

UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
INSTITUTE OF COMPUTER SCIENCE

Rein Raudjärv

Dynamic Schema-Based Web Forms Generation in Java

Master Thesis (30 EAP)

Supervisor: Marlon Dumas, *PhD*

Autor: “.....” mai 2010

Juhendaja: “.....” mai 2010

Lubada kaitsmisele

Professor: “.....” 2010

TARTU 2010

Contents

Chapter 1 Introduction	4
Chapter 2 Requirements	8
2.1. Form Generator	8
2.2. Supported Subset of the XML Schema Language	9
2.3. DynaData	9
Chapter 3 State of the Art	12
3.1. Form Generators	12
3.2. XForms	14
3.3. XForms Rendering	15
Chapter 4 High-Level Architecture	17
4.1. Overview	17
4.2. Form Components	17
4.3. Form Elements	19
4.4. Layout	21
4.5. DynaData	24
Chapter 5 Walkthrough	28
5.1. Sample Application	28
5.2. Schema and Form	28
5.3. Writing and Reading XML	30
5.4. Customization	33
5.5. Schema Changes	35
Conclusions	39
Summary (in Estonian)	41
Bibliography	43
Appendix 1 Implementation Details	45
1. Overview	45
1. Form	46
2. Writing and Reading XML	55
3. Rendering Form with Aranea	65
4. DynaData	71
Appendix 2 CD with the Source Code	73

Chapter 1

Introduction

The problem area of this thesis is that of **automated generation of web forms**.

A **web form** on a web page or a fragment of the web page allows a user to enter data that is sent to a server for processing. Traditionally each **web form is developed manually** by instantiating and combining a set of standard form controls like text fields, radio buttons, checkboxes etc. This could be achieved by using a visual tool or a code editor. This manual approach to form development process gives high degrees of flexibility, since the developer has full control over the structure and look-and-feel of the form. However, it is also a time-consuming process, especially when the form is expected to change frequently. In Java [GJ00] this process could mean writing a custom JSP (Java Server Pages) page [Be03] for displaying the form and a Java Servlet [HC01] for processing the posted data. In case the service layer is changed and we have to add a new field to the form we must manually update the JSP and Java Servlet, recompile and redeploy the application to the server.

Instead of developing a web form manually we could also use a **Domain-Specific Language (DSL)** [DK00] to specify the web form. Given a specification of the web form in this DSL, we could then generate the form. Since the structure of the web form is highly dependent on the structure of the data to be collected by the form, it makes sense to take as a basis, a language that can be used to capture the schema (or type) of a document. To describe a type of web document **XML Schema Definition (XSD)** [Wa02] is the most widespread and expressive language. In case the web form is processed by an XML web service [Ce02] we could actually have the XML schema already. Otherwise it has to be written manually. Although the XML schema is powerful it is meant for describing the type of XML [Ra03] data not how to represent it to a human. E.g. it says that *first name* is a String and *birth date* is a date but not whether to display them beside or below each other or which exact user interface controls to use. To describe the presentation aspects, we could add annotations to the corresponding schema elements. This requires that the original XML schema has to be updated.

There are many tools available both for designing a form visually and generating a web form based on an XML instance or an XML schema. These tools can save us time but they expect that the form is constant and does not change. In case the original XML schema changes we have to manually compare it with the annotated schema, find which parts were changed and update the latter. In worst case we have to go back and run the visual tools to design the form from scratch. Then we must regenerate the Java code for the web form and redeploy it. This approach leads to a downtime, during which the web form is not available – unless we continued to use the old version of the form, but then we would not be collecting data according to the new schema, leading to data inconsistency.

The motivation of this thesis is to avoid or mitigate this downtime problem. Whenever the schema changes, we would like to reduce the time and the number of steps required to update the web form. Instead of running the form generating tools manually we could **generate a web form entirely at runtime**, based on the available presentation annotations and without requiring additional input from the form developer.

This would eliminate the need for form redesign, code compilation and redeployment. The crux of this approach is that the **presentation annotations are optional**. If we were using plain XML schema without annotations we would always get an up to date web form with zero downtime in case the schema changes. In contrast, if we mandate that the presentation annotations must be updated every time that the schema changes, then the form would not be updated until a form developer manually changed the presentation annotations after a schema change.

To streamline even more the change process, we propose to **keep the annotations in a separate file** from the XML schema and **not assume they are always up to date with the schema**. This allows us to always display an up to date web form. In case the schema changed we apply the old annotations as much as possible and use some conventions for displaying the parts of the form not covered with the annotations. Until someone has updated the annotations we get a web form that is technically up to date but may not display the changed parts in the best way. In other words, the approach we take is based on the concept of “**graceful degradation of user interfaces**” [FV04]: when the schema changes, the form immediately becomes available under the new schema, with a possible degradation of the quality of the presentation, until the presentation metadata is updated by the form developer.

To further reduce the amount of time and effort needed to maintain the presentation annotations up-to-date, we adopt a number of default conventions for **generating the presentation annotations** for each type of XML schema element. When the schema changes, we can then **automatically update the annotations** made inconsistent by the update. We identify the new schema parts and generate the annotations marked as new. Also if some parts were removed from the schema we group these annotations together which could not be applied to the form. By isolating the changes in the annotations the form developer is relieved from the task of manually comparing the old and the new schema in order to determine which presentation annotations need to be added, deleted or modified. Also, if the new annotations are generated automatically, the form developer only needs to modify them to reflect his/her intentions, instead of writing them from scratch.

In summary, the aim of this thesis is to design and implement a **web form generator** producing forms out of XML schemas and presentation specifications which are allowed to evolve independently. The presentation specification captures details like for example a particular form element must be rendered as a list box or the contents of a given element should be displayed as a list or as a table. The default presentation specification should be generated automatically out of XML schema. The layout choices captured in the existing presentation annotations should be preserved as far as possible when the XML schema changes.

Our web form generator implementation is called **DynaForm**. It is a reusable web component based on **Aranea Web Framework** [MK06] although it is to a large extent framework-independent and it could be extended later on to support other Web frameworks, such as Spring MVC [LD06] or Java Server Faces [Be04]. The component takes an XML schema, an optional presentation specification written in a language called **DynaData** and an optional XML instance as input and provides a generated web form, a generated/updated DynaData file and an XML instance filled with form data as output.

Prerequisites

The thesis assumes that the reader is familiar with the concept of web form and with XML technology. Basic knowledge of XML schema and some knowledge of Java programming are assumed. For understanding the web implementation part some

knowledge of **Aranea Web Framework** is needed. For a quick introduction see the introductory tutorial [Ka10].

Contributions and Outline

The thesis is divided into five chapters and two appendixes.

This is the first chapter which introduces the problem of the thesis.

The second chapter describes the functional and non-functional requirements of the web form generator **DynaForm** and its **DynaData** language.

The third chapter covers alternative web form generators available as well as XForms and its associated rendering tools.

The fourth chapter describes the DynaForm architecture and DynaData language which allows customizing the generated web forms.

The fifth and final chapter provides the usage scenario of the tool developed in this work by showing how changes to the XML schema are handled.

The first appendix describes the implementation details of DynaForm.

The second appendix is a CD containing the source code of the tool.

The main contribution of this thesis is the **runtime web form generator** called **DynaForm**. The distribution is available on the accompanying CD and from the website: <http://code.google.com/p/xsd-web-forms/>.

Chapter 2

Requirements

This chapter describes the functional and non-functional requirements of the web form generator DynaForm and its DynaData language.

2.1. Form Generator

The inputs and outputs of DynaForm are shown on the figure 2.1.

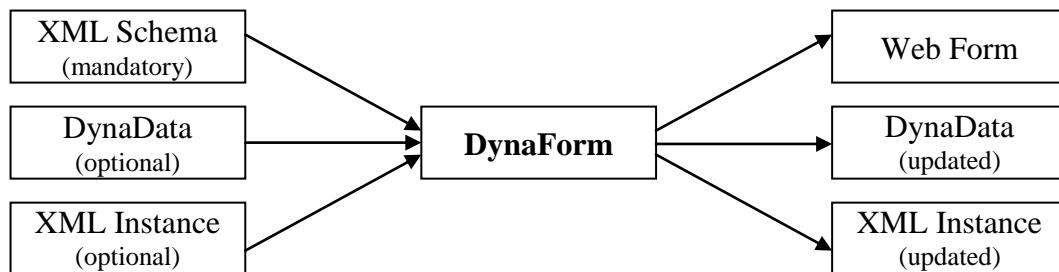


Figure 2.1. DynaForm input and output.

The inputs of DynaForm are the following:

1. **XML schema** provides the structure of the form.
2. An optional presentation specification in a language called **DynaData**, which provides custom metadata for rendering the form. If omitted default rendering conventions are used.
3. An optional **XML instance** provides the initial values of the form. If omitted a form is generated without any pre-populated values.

The outputs of DynaForm are the following:

1. The main output is the **HTML [MK96] web form** itself. A certain set of CSS classes are used so that the layout can be further configured also via a custom CSS.
2. New **DynaData** is automatically generated. If the input DynaData was not provided, this just provides the values based on default rendering conventions. So we can easily customize the form rendering by just altering some of the DynaData values.
3. An **XML instance** resulting from the input data provided by the user is produced when the form is submitted.

We continue by defining the set of XML schema features DynaForm should support.

2.2. Supported Subset of the XML Schema Language

An XML schema can be very complex. To allow an XML instance to be extended an XML schema may even declare that an element may contain any set of sub-elements. Covering all XSD features is out of the scope of this thesis. Accordingly, we have chosen to focus on a representative subset of XML Schema Definition (XSD) elements.

The supported subset of XSD should fulfill the following requirements:

1. The set of XSD elements allowed in the input of DynaForm are:
 1. `<all>`,
 2. `<attribute>`,
 3. `<choice>`,
 4. `<element>`,
 5. `<sequence>`.
2. Reoccurring XSD elements (`minOccurs` and `maxOccurs` attributes) are also allowed.
3. The supported XSD simple data types are:
 1. String data types: `string`.
 2. Date data types: `date`, `time`, `datetime`.
 3. Numeric data types: `integer`, `long`, `int`, `short`, `byte`, `decimal`.
 4. Miscellaneous data types: `boolean`.

We now proceed by defining the requirements for the DynaData language.

2.3. DynaData

DynaForm enables developers to customize a generated web form using a presentation specification language called **DynaData**.

The DynaData presentation specification language should fulfill the following functional requirements:

1. A DynaData specification should be composed of independent elements. Each DynaData element should capture an assertion about how a given XML schema element should be rendered. Thus, each metadata element should refer to the XML schema element it describes.

2. Each DynaData element that refers to a schema `<element>` of simple type or a schema `<attribute>` should describe the following information:
 1. **Label** – a short text to be displayed aside this field.
 2. **Control** – a user interface component that displays and allows changing the current value (the minimum set is described below).
 3. **Read only flag** – if enabled user cannot alter the field value.
 4. **Required flag** – if enabled user is required to enter a value.
3. Each DynaData element that refers to a schema `<element>` of complex type should describe the following information:
 1. **Label** – a short text to be displayed above the sub content.
4. Additionally DynaData elements should be grouped into sections.

A DynaData element maps an XML schema element to one of the following User Interface (UI) controls:

1. **Input** – a free-form data entry or a user interface component appropriate to the current data type (e.g. text box for a string, check box for a boolean etc).
2. **Secret** – a password entry (value can be entered but is not displayed).
3. **Text Area** – a multi row free form data entry.
4. **Output** – a form control that renders the current value but it provides no means for entering or changing data.
5. **Select One** – a form control that allows a user to make a single selection from multiple choices.

The non-functional requirements of the DynaData language are:

1. The syntax of the DynaData must be concise.
2. The syntax of the DynaData must be easy to learn for developers.

If omitted DynaData must be generated automatically and if the schema changes new generated values should overwrite the existing values unless they are actually altered by the developer. To distinguish the customized values from the generated ones we divide the DynaData file into sections. Thus the developer can edit the generated values but he/she must move the custom lines into the correct part of the file. Otherwise the custom values will be overwritten next time the DynaData is synchronized with the schema.

The functional requirements for the DynaData processing are:

1. DynaForm collects DynaData from a *Form* – the generated *Form* is used to collect default values for each form component.
2. DynaForm reads custom DynaData from a file – a DynaData file may customize any form component.
 1. The DynaData file is optional and it may be partial. The generated *Form* provides default values for any component missing from the file.
 2. The DynaData file may be invalid by including form components which do not belong to the *Form*. This may happen if the corresponding element was removed from the schema.
 3. The DynaData file is divided into sections based on the classification described below. The *Custom* and *Broken* DynaData is read, the *Generated* DynaData is ignored.
3. DynaForm classifies DynaData that was read into the following sections:
 1. ***Custom*** – values describing form components included both in the file and in the form.
 2. ***Broken*** – values describing form components included in the file but not found in the form. Probably because the corresponding elements were removed from the XML schema.
 3. ***Generated*** – values describing form components included in the form but not found in the file (*Generated* section ignored). This includes all values if the file was omitted.
4. DynaForm applies *Custom* DynaData to a form – updates the form based on the values read from a file.
5. DynaForm writes DynaData to a file – writes values read from a file and/or collected from the form.
 1. The file is divided into sections based on the classification described above.
 2. If the file is missing it is created based on only the values collected from the form. Otherwise the existing file is preserved except the *Generated* section. The latter is always regenerated and it complements the custom values read from other sections. The whole DynaData file written always covers all form components.

Chapter 3

State of the Art

In this chapter we will give a short overview of available web form generators, the **XForms** format and tools for rendering it.

3.1. Form Generators

When comparing DynaForm with alternative form generators we consider the following evaluation criteria:

1. **Universality** – a form generator must be generic by taking any XML schema as input.
2. **Completeness** – a form generator must support as many features of the input as possible.
3. **Customization** – the web form presentation must be customizable.
4. **Evolution** – any customization should survive the changes to the input schema as much as possible.

The following form generators could be found from the web:

1. XML Form Generator by Lea Smart (created in ca 2000)

This is an ASP script for generating questionnaire HTML forms. It takes an XML describing the form fields as input and generates an HTML form as output. Each form is a list of question-answer pairs so basically form fields with labels. Most HTML input controls are supported. The generated form includes client-side validation. The tool can be used online at

<http://www.totallysmartit.com/examples/xml/questionnaire/>.

This is not a generic form generator as its limited to very simple forms.

2. XML Form Generator by Andrew Mooeny (2004)

This is another simple HTML form generator written in ASP.NET. It only allows using a single list of form controls. There is also another script which produces a GRID for editing the form fields. An article with online demos is available at:

<http://aspalliance.com/488>.

This is also not a generic form generator as its limited to very simple forms.

3. HTML Form Generator from XML by Ed Lai (started 2004, last update in 2006)

This is a Perl script which takes an XML instance or an XML schema as input and produces an HTML form as output. An incomplete set of XSD features is supported. The generated form applies some validation rules to the data but there is no support for adding/removing elements for dynamic lists. The form can be submitted to another script which converts the data into an XML document. The tool can be used online at <http://www.datamech.com/XMLForm/>.

This is a generic form generator as it takes an XML instance or a schema as input. Nevertheless comparing to DynaForm it supports less XSD features and has no support for customization other than applying a custom CSS or transforming the HTML via XSLT (for example).

4. Xydra – Form Generator for Web Services by Extreme Computing Lab (2003)

This is a Java Servlet which takes an XSD schema or a WSDL as input and produces an HTML form as output. If the form is submitted to the server an XML document is constructed and sent to the Web Service. There is no support for adding/removing elements for dynamic lists. To customize the look and feel the same HTML that is generated can be edited and then used as the template for the Servlet. The library is written in Java 1.4 with additional JSR14 generics enhancements.

The project is hosted at <http://www.extreme.indiana.edu/xgws/xydra/>.

This is another generic form generator. It supports customization via templates which could be still used even if the schema changes. Comparing to DynaForm it supports less XSD features and it is discontinued for 7 years.

5. XML Forms Generator by IBM (started 2005, last update in 2009)

This is an Eclipse plug-in that takes an XML instance or a WSDL as input and produces XHTML with XForms [Du03] embedded as output. The result could be customized using visual form builders. To view the form a browser with XForms support is required. Alternatively additional tools could be used to transform XForms into regular HTML + JavaScript [FI06].

The project is hosted at <http://www.alphaworks.ibm.com/tech/xfg>.

This is a generic form generator as it takes an XML instance or a WSDL as input. Comparing to DynaForm it does not produce a regular HTML but rather XForms embedded into XHTML. After generation the form can be customized manually or by separate tools. In either case there is no support for schema evolution. If the input happens to change the output must be manually regenerated and any customization to the output is lost.

To sum up, none of the DynaForm alternatives fulfill the same aspects. They are either not generic at all, do not support advanced features such as dynamic lists or the customization does not survive the schema changes.

DynaForm produces regular HTML + JavaScript directly which makes it lightweight. However we could also output XForms as the IBM XML Forms Generator does, and then convert it into HTML using a third party tool. We continue by shortly describing the XForms format and then provide a list of tools for rendering it.

3.2. XForms

XForms is an XML format for the specification of a data processing model for XML data and user interface(s) for the XML data, such as web forms. XForms was designed to be the next generation of HTML / XHTML forms, but is generic enough that it can also be used in a standalone manner or with presentation languages other than XHTML to describe a user interface and a set of common data manipulation tasks.

An example of XForms:

```
<xforms>

  <model>
    <instance>
      <person>
        <fname/>
        <lname/>
      </person>
    </instance>
    <submission id="form1" method="post" action="submit.html"/>
  </model>

  <input ref="fname">
  <label>First Name</label></input><br />

  <input ref="lname">
  <label>Last Name</label></input><br /><br />

  <submit submission="form1">
  <label>Submit</label></submit>

</xforms>
```

Unlike an HTML form with XForms, input data is described in two different parts:

- **The XForm model** – defines what the form is, what it should do, what data it contains. XForms is always **collecting data for an XML document**. The instance element in the XForms model defines the XML document.

- **The XForm user interface** – defines the input fields and how they should be displayed. The user interface elements are called controls (or input controls). In the example above the two `<input>` elements define two input fields. The `ref="fname"` and `ref="lname"` attributes point to the `<fname>` and `<lname>` elements in the XForms model. The `<submit>` element has a `submission="form1"` attribute which refers to the `<submission>` element in the XForms model. A submit element is usually displayed as a button. Notice the `<label>` elements in the example. With XForms every input control element has a required `<label>` element.

XForms is not designed to work alone. There is no such thing as an XForms document. XForms has to run inside another XML document. It could run inside XHTML 1.0, and it will run inside XHTML 2.0.

We proceed by describing some of the tools that enable to render XForms.

3.3. XForms Rendering

The XForms rendering tools can be divided into the following categories:

1. Browser plug-ins that enable to render XForms embedded into HTML pages.
2. JavaScript libraries which translate XForms into regular HTML + JavaScript.
3. XSLT which could be used both client-side and server-side.
4. Server-side libraries which translate XForms into regular HTML + JavaScript.

We continue by examining these categories by listing the corresponding tools.

Browser plug-ins that enable to render XForms embedded into HTML pages are:

1. **Mozilla XForms** (<http://www.mozilla.org/projects/xforms/>) – Mozilla Firefox extension for rendering XForms. Full XForms support is incomplete. Latest release on October 7th 2008.
2. **formsPlayer** (<http://www.formsplayer.com/>) – Internet Explorer extension for rendering XForms. Full XForms standard supported. Latest release on January 11th 2008.

JavaScript libraries which translate XForms into regular HTML + JavaScript are:

3. **FormFaces** (<http://www.formfaces.com/>) – a JavaScript processor which translates XForms into regular HTML. Supports most of the browsers. Latest release in 2007.

4. **Ubiquity XForms** (<http://code.google.com/p/ubiquity-xforms/>) – a processor that allows developers to use XForms markup to create interactive web applications. Ubiquity XForms adds new APIs to a number of popular AJAX [Ho08] libraries, making XForms processing available in standard browsers, without the need for a download. It is available as open source. Latest release on January 25th 2010.
5. **EMC Documentum XForms Engine** (codenamed **Formula**) (<https://community.emc.com/docs/DOC-4345>) is an XForms 1.1 engine based on the Google Web Toolkit [SB08]. It is coded in the Java programming language, and compiled into JavaScript which can be executed by most modern browsers.

XSLT which could be used both client-side and server-side is:

6. **XSLTForms** (<http://sourceforge.net/projects/xsltforms/>) – XForms to XHTML + JavaScript (AJAX) conversion based on a unique XSL transformation [Ti01]. Suitable server-side (PHP [LT06]) or client-side (Internet Explorer, Mozilla Firefox, Opera, Safari) browser treatment where an XSLT 1.0 engine is available. Latest release on October 10th 2009.

Server-side libraries which translate XForms into regular HTML + JavaScript are:

7. **betterFORM** (<http://www.betterform.de/>) (open source license) Latest release on March 5th 2010.
8. **Chiba** (<http://chiba.sourceforge.net/>) (open source license) (Latest release is Chiba Web 3.0.0b2 from June 25th, 2009)
9. **Orbeon Forms** (<http://www.orbeon.com/>) (open source license) Includes online form builder. (Latest release is 3.7.1 from June 2nd, 2009)

This finishes our overview of alternative tools. In the next chapter we will continue by describing the architecture of DynaForm.

Chapter 4

High-Level Architecture

This chapter describes the main contribution of this thesis.



4.1. Overview

Our web form generator implementation is called **DynaForm**. It is a reusable web component based on **Aranea Web Framework** although support for other web frameworks could be added later. The component takes an XML schema, an optional DynaData and an optional XML instance as input and provides a generated web form, a generated/updated DynaData file and an XML instance filled with form data as output (see chapter 2.1 on page 8).

As an XML schema defines a hierarchical data structure our generated form is also a **hierarchical** composition. Each form component stores some data about the schema element it represents and encodes certain rules about how it should be displayed. Based on the types of schema elements we have also provided the corresponding types of form components. We continue by listing and shortly describing each of them.

4.2. Form Components

The *Form* component types with the corresponding XML schema elements are listed in table 4.1.

Form Component	XML Schema Element	Example	Description
<i>Form Element</i>	<element> of simple type or <attribute>		A single form field with label and value of a certain type.
<i>Form Section</i>	<element> of complex type		A component wrapper adding a label, no value of its own.


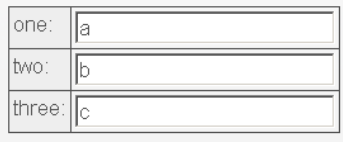
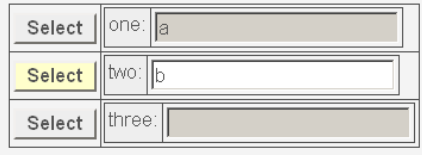
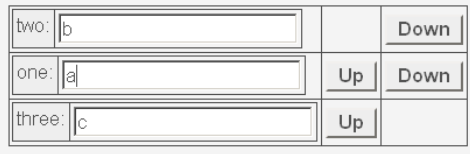
Form Repeat	Any element with <code>maxOccurs > 1</code>		Dynamic list of sub-components of same type. Can have minimum and maximum limits.
Form Sequence	<code><sequence></code> or content of <code><element></code> of complex type		Static list of sub-components.
Form Choice	<code><choice></code>		Static list of sub-components, one of which is selected.
Form All	<code><all></code>		Reorderable list of sub-components, any of which can be enabled or disabled.

Table 4.1. Form component types with the corresponding XML schema elements.

A *Form Element* represents a schema `<element>` of simple type or an `<attribute>`. It is a labeled single form field displaying a current value and allowing user to alter it. *Form Elements* can be thought as the leaves and rest of the components as branches of the form structure. Comparing the form with schema structure there is no difference between an `<element>` of simple type and an `<attribute>` – this is only relevant when writing or reading an XML instance.

A *Form Section* represents a schema `<element>` of complex type. It contains another *Form* as the content plus a label. If the complex `<element>` consists only of a single attribute or has no attributes at all the *Form Section* is constructed with the corresponding form component as the content. Otherwise a synthetic *Form Sequence* is used to group together all attributes and the body.

A *Form Repeat* represents a repetition of schema element (`maxOccurs > 1`). It is also used for optional elements (`minOccurs = 0` and `maxOccurs = 1`). A *Form Repeat* contains a dynamic number of instances of same sub-component.

A *Form Sequence* represents a schema `<sequence>`. It is also used to group together XML attributes and an XML element body. It contains a fixed list of sub-components each of which may be of different type.

A *Form Choice* represents a schema `<choice>`. It always contains a fixed list of sub-components each of which may be of different type. In XML exactly one of the sub-components must occur. So in the *Form* user must be able to select this component.

A *Form All* represents a schema `<all>`. It always contains a fixed list of sub-components each of which may be of different type. In XML any of the sub-components may occur but not more than once. The sub-components may occur in any order. So in the *Form* user must be able to enable or disable each component as well as choose their order.

We continue by describing the *Form Element* in detail.

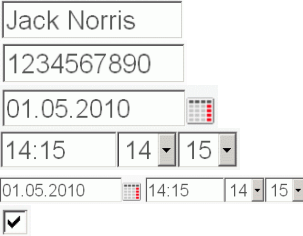
4.3. Form Elements

To recall a *Form Element* represents a schema `<element>` of simple type or an `<attribute>`. It is a labeled single form field displaying a current value and allowing user to alter it.

A *Form Element* has the following properties:

1. **Data type** – a set of possible values assigned to the form field.
2. **Value** – a current value of this field.
3. **Label** – a short text to be displayed aside this field.
4. **Control** – a user interface component that displays and allows changing the current value.
5. **Possible restrictions** – some rules narrowing the set of possible values assigned to this form field.
6. **Possible read only flag** – if enabled user cannot alter the field value.
7. **Possible required flag** – if enabled user is required to enter a value.

The available UI *Controls* for *Form Elements* are listed in table 4.2.

Control	Examples	Description	Attributes
<i>Input</i>		A free-form data entry or a user interface component appropriate to the current data type.	-





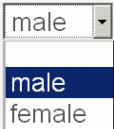
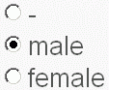
Text		A free-form text entry.	size – width in characters
Secret		A password entry.	size – width in characters
TextArea		A multi row free-form text entry.	cols – width in characters, rows – height characters
Output	Jack Norris 1234567890 01.05.2010 14:15 01.05.2010 14:15 x	A form control that renders the current value but it provides no means for entering or changing data.	-
SelectBox		A list box that allows user to select a single row.	size – height in lines
ComboBox (also <i>SelectOne</i>)		A drop down that allows user to select a single row.	-
RadioSelectBox		A list of radio boxes that allows user to select a single item.	-

Table 4.2. The list of available UI Controls for Form Elements.

The possible restrictions for a *Form Element* are:

1. An enumeration of possible values.
2. The exact length of value.
3. The minimum length of value.
4. The maximum length of value.
5. The minimum inclusive value.
6. The minimum exclusive value.
7. The maximum inclusive value.
8. The maximum exclusive value.

This finishes the overview of the *Form Element*. As of other components we only focus on a DynaForm-specific aspect such as layout. We continue by describing the layout options for *Form Section* as well as *Form Repeat* components.

4.4. Layout

One aspect of a web form is its layout – do the form fields appear below each other, side by side, in columns, as a table etc. Our goal was to be able to render a table so that each *Form Repeat* element would draw up a row. Anything more than that was considered to be outside the scope of this thesis.

As the result we composed two new properties for describing the layout – ***Section Style*** for the *Form Section* and ***Repeat Style*** for the *Form Repeat*. We continue by describing both of them.

4.4.1. Section Style

We wanted to configure **rendering direction** of form components – either show components below each other (**vertically**) or side by side (**horizontally**). As it always involves a set of components the rendering direction is not a property of a single *Form Element* but rather a component container like *Form Section*, *Form Repeat*, *Form Sequence*, *Form Choice* or *Form All*. Using so many different components is complicated. Instead we assigned the rendering direction property only to the *Form Section*.

As the property can be configured only using DynaData, referring to a *Form Section* is easy as each *Form Section* corresponds to a certain schema `<element>` which is always given a name. Other containers listed do not have any names.

Although a *Form Section* itself always contains a single sub-component we designed the rendering direction to be inherited by all sub-components. Considering rendering direction we think of each *Form Section* as a level of scope. Other containers (we inspect *Form Repeat* later) get their direction from the direct parent *Form Section*. If no direction is assigned to a *Form Section* it also inherits the property from its parent *Form Section*. A *Form Section* can be assigned a rendering direction either **locally** (affecting only successor components under the same *Form Section*) or **fully** (affecting all successor components).

We name the property of *Form Section* rendering direction as ***Section Style***. Using two directions, two levels of impact and unassigned value we get five different values as shown in table 4.3.

Section Style	Direction of direct children	Direction of other successors
<i>Inherit</i> (default)	(inherited from the parent)	(inherited from the parent)
<i>Row</i>	horizontal	(inherited from the parent)
<i>Column</i>	vertical	(inherited from the parent)
<i>Horizontal</i>	horizontal	horizontal
<i>Vertical</i>	vertical	vertical

Table 4.3. The list of all *Section Styles*.

So *Horizontal* and *Vertical* values will be inherited by all sub-components whereas *Row* and *Column* will affect only the sub-components directly under the given *Form Section*. The parent *Section Style* for the root *Form Section* is always *Vertical*.

Combining the *Section Style* values we get four ways for rendering *Form Sections* inside another *Form Section* as shown in table 4.4.

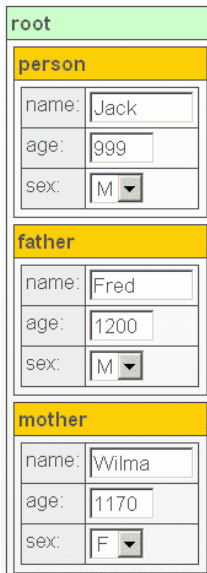
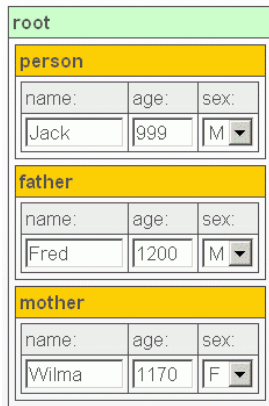

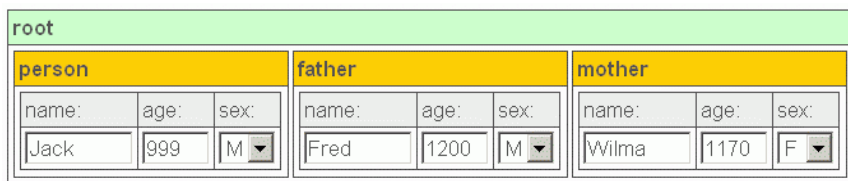
<p><i>Vertical</i> (also <i>Column</i> inside <i>Vertical</i>)</p> 	<p><i>Column</i> inside <i>Horizontal</i></p> 	<p><i>Row</i> inside <i>Vertical</i></p> 
<p><i>Horizontal</i> (also <i>Row</i> inside <i>Horizontal</i>)</p> 		

Table 4.4. The examples of *Section Style* values considering their parent *Section Style*.

As we can see there are two examples where “person”, “father” and “mother” are drawn below each other and two versions where they are beside each other. At the same there are two options for the “name”, “age” and “sex” direction.

Also notice that when *Form Elements* are drawn vertically the labels are kept on the left, otherwise the labels are automatically put on the top.

4.4.2. Repeat Style

Suppose we have a *Form Repeat* (think it as a table) containing a dynamic number of *Form Sequences* (table rows) each of which consists of a certain set of *Form Elements* (table cells). We want to render this composition as a table – *Form Sequences* below each other and *Form Elements* beside each other. Assuming we also have a parent *Form Section*, using *Section Style* alone allows us to choose one direction for the whole structure – this is clearly not enough.

For the *Form Repeat* we need another rendering direction property which would allow us to choose the directions for *Form Sequences* as well as their sub-components. Combining these directions we will get the three (we exclude the last one which is useless) values for the ***Repeat Style*** as shown in table 4.5:

Repeat Style	Direction of <i>Form Sequence</i>	Direction of sub-elements
Normal (default)	vertical	vertical
Rows	vertical	horizontal
Columns	horizontal	vertical
(Not used)	horizontal	horizontal

Table 4.5. The list of *Repeat Styles*.

Normal

Rows

Columns

Table 4.6. The examples of *Repeat Styles*.

We excluded the last combination as it draws everything in a single row which is very hard to catch for an eye. The rest of the options are illustrated in table 4.6.

root				
#	name	age	sex	
1	Jack	999	M	Remove
2	Fred	1200	M	Remove
3	Wilma	1170	F	Remove
Add				

Figure 4.1. A Table Repeat Style

Rows looks almost like a table except the column headers are repeated for each row. Based on that we finally added the **Table** option which extracts the labels from each *Form Sequence* sub-component and renders them in a single heading row skipping them inside the table (see figure 4.1).

This ends our overview of layout options. Next we carry on by describing the DynaData language in detail.

4.5. DynaData

DynaForm enables developer to customize a generated web form using a presentation specification language called **DynaData** (see chapter 2.3 on page 9 for the requirements).

In this chapter we will describe a sample DynaData file and then examine the syntax of the DynaData in general.

4.5.1. Sample DynaData

The sample DynaData file follows:

```
@CUSTOMIZED
//root { label: "Root"; layout: row; }
//name { label: "Name"; readonly: false; required: false; control:
Secret; }
//married { label: "Married"; readonly: true; required: false; control:
Input; }
//age { label: "Age"; readonly: false; required: true; control: Input; }

@BROKEN

@GENERATED
```

The sample describes four form components – `root`, `name`, `married` and `age`. The set of attributes used to describe each one depends on the type of the form component. In this case `root` is a *Form Section*, other components are *Form Elements*. Each one is given a custom label (in this case just the name with a first letter in upper case). The `root`

component is assigned a `row layout` which states a *Section Style* where all sub-components are rendered in a row. Other components are attached with a `readonly` value, a `required` value and a *Control*. If `readonly` is set as `true` the value of the form component cannot be changed. If `required` is set as `true` the value of the form component cannot be leaved blank.

4.5.2. DynaData Syntax

The syntax of the DynaData file is based on the **Cascading Style Sheets (CSS)** [Me06] **file format** which is both concise and already familiar to web developers. Each line contains a *selector* and a group of *declarations*. In addition DynaData lines are grouped into *sections*:

```
file := section
      section
      ...

section := @section_title
          line
          line
          ...

line := selector { declaration declaration declaration ... }
declaration := attribute: value;
```

A *selector* chooses a form component to describe. A *declaration* describes it. Each declaration is a key-value pair assigning a value to a particular attribute of the form component described. The file is divided into *sections* to distinguish generated lines from the ones manually customized. In the next chapters we continue by describing *selectors*, *attributes* and *sections* each of them in details.

4.5.3. DynaData Selectors

Each line of the DynaData file (except the *section* marker) starts with a *selector*. It chooses one or more form components the corresponding line describes. There are two kinds of selectors supported:

- **General selector** – e.g. `//city` – a name of a form component,
- **Full selector** – e.g. `/shiporder/shipto/city` – a full path to a form component.

The *general selector* chooses all form components with the given name.

The *full selector* chooses form components by naming a root level component, a second level component, a third level component etc – the whole path. Out of all

component types only *Form Elements* and *Form Sections* have names. Therefore each path item is a *Form Section* except the last one which can be either a *Form Section* or a *Form Element*.

To keep the DynaData file short only *general selectors* should be used as much as possible. *Full Selectors* should only be used in case the same component occurs in many places and each occurrence needs to be configured differently.

4.5.4. DynaData Attributes

Each *selector* is followed by a set of *declarations*. Each declaration is a key-value pair assigning a value to a particular attribute of the form component described. The valid set of attributes is defined by the type of the form component described.

The supported attributes for each type of form component are shown in table 4.7:

Form Component	Attribute	Type	Description
<i>Form Element</i>	label	string	A short text to be displayed aside this field.
	readonly	true/false	If enabled user cannot alter the field value.
	required	true/false	If enabled user is required to enter a value.
	control	<i>Control</i>	A user interface component that displays and allows changing the current value (see chapter 4.3 on page 19 for the list of <i>Controls</i>).
	size	integer	A width (in characters) or a number of lines of the <i>Control</i> if supported.
	cols	integer	A width (in characters) of the <i>TextArea</i> .
	rows	integer	A height (in characters) of the <i>TextArea</i> .
<i>Form Section</i>	label	string	A short text to be displayed above the sub content.
	layout	<i>Section Style</i>	A rendering direction for the sub-components. The value must be one of the following: <code>inherit</code> <code>row</code> <code>column</code> <code>horizontal</code> <code>vertical</code> . (See chapter 4.4.1 on page 21 for the descriptions.)

<i>Form Repeat</i>	repeat-style	<i>Repeat Style</i>	A rendering direction for the repeated sub-components. The value must be one of the following: <code>normal</code> <code>rows</code> <code>columns</code> <code>table</code> . (See chapter 4.4.2 on page 23 for the descriptions.)
---------------------------	--------------	----------------------------	---

Table 4.7. The list of supported DynaData attributes.

4.5.5. DynaData Sections

As we already described in the requirements on page 9 the DynaData file is divided into three *sections* (see table 4.8).

Section	Description
<i>Custom</i>	This is the section containing all correct custom descriptions of the form components. These are the ones from <i>Custom</i> and <i>Broken</i> sections that were also found in the schema.
<i>Broken</i>	This is the section containing all broken custom descriptions of the form components. These are the ones from <i>Custom</i> and <i>Broken</i> sections that were not found in the schema.
<i>Generated</i>	This is the section containing all correct non-custom descriptions of the form components. These are the ones that were not found in <i>Custom</i> or <i>Broken</i> section but were found in the schema.

Table 4.8. The list of DynaData sections.

The *Generated* section always complements the *Custom* and *Broken* sections. When a DynaData file is first generated it only contains the *Generated* section as other sections are first empty. On the other hand if the *Custom* and *Broken* sections cover all form components the *Generated* section will be empty.

It is important to keep in mind that the *Generated* section is overwritten every time DynaForm processes the DynaData. Therefore when changing a line in that section it must also be moved to the *Custom* section.

This completes our discussion about the DynaData language and also our overview of the high-level architecture of the DynaForm. In the next chapter we will take a sample schema and see the DynaForm in action.

Chapter 5

Walkthrough

This chapter demonstrates the web form generator using a sample schema.

5.1. Sample Application

To demonstrate the web form generation we have also developed a **sample application** which enables a user to try the form generation with a set of XML schemas. After selecting a schema user can switch between the following modes:

1. **XSD** – view the schema source.
2. **Form** – view the generated form and fill it with data.
3. **XML** – view and edit the generated XML instance.
4. **DynaData** (also “**Metadata**”) – view and edit the generated DynaData file.

To view and edit the form data user can use both *form mode* and *XML mode* and switch between them when necessary.

We continue by generating a web form based on a sample schema. We will show that the data can be edited both in *form mode* and *XML mode*. Then we will customize the form by altering the automatically generated DynaData. Finally we will see what happens when the original schema changes.

5.2. Schema and Form

We use the following sample XML schema:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

```

        </xs:complexType>
    </xs:element>
    <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="title" type="xs:string"/>
                <xs:element name="note" type="xs:string" minOccurs="0"/>
                <xs:element name="quantity" type="xs:positiveInteger"/>
                <xs:element name="price" type="xs:decimal"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:sequence>
<xs:attribute name="orderid" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>

</xs:schema>

```

The schema defines a root element "sihporder" which has an attribute "orderid" and the following sub elements: an "orderperson", a "shipto" and at least one "item" element. The "shipto" and "item" elements have corresponding sub elements.

An empty generated form is shown on figure 5.1.

Figure 5.1. An empty sample form.

The navigation buttons on top are part of the sample application and do not belong to the generated form.

As we see the form contains all elements and attributes defined in the schema. Each simple element or attribute corresponds to a form field (*Form Element*) and each complex element corresponds to a titled box in the form – a *Form Section*. By default names of the defined elements and attributes are used as labels.

As the “item” element was defined with unbounded maximum occurrence the form contains an “Add item” button and a “Remove” button beside each “item” if there are at least two of them. The “note” element was defined with zero minimum occurrences so it is omitted unless an “Add note” button is pressed.

We continue by filling the form with data and producing an XML instance out of it.

5.3. Writing and Reading XML

Assume we have filled the form as shown on figure 5.2.

Schema: example1.xsd		Choose	Reload	Delete metadata
XSD	Form	XML	Metadata	Save
		Reset	<input checked="" type="checkbox"/> Auto save	
shiporder				
* orderid:	325472			
orderperson:	Jaanika Tamm			
shipto				
name:	Jaan Tamm			
address:	Pärna 40			
city:	Tartu			
country:	Estonia			
item				
1	title: Madonna - Erotica			
	Add note			
	quantity: 1			
	price: 10.90			
Add item				

Figure 5.2. A filled sample form.

By pressing the “XML” button we can switch to the *XML mode* and the form data is used to produce a nice indented XML document (see figure 5.3).

Schema: example1.xsd Choose Reload Delete metadata

XSD Form XML Metadata Save Reset

```
<shiporder orderid="325472">
  <orderperson>Jaanika Tamm</orderperson>
  <shipto>
    <name>Jaan Tamm</name>
    <address>Pärna 40</address>
    <city>Tartu</city>
    <country>Estonia</country>
  </shipto>
  <item>
    <title>Madonna - Erotica</title>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
</shiporder>
```

Figure 5.3. An XML instance produced from the data filled in the sample form.

We can change the data in *XML mode* for example (see figure 5.4).

Schema: example1.xsd Choose Reload Delete metadata

XSD Form XML Metadata Save Reset

```
<shiporder orderid="889923">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

Figure 5.4. A modified XML instance of the sample form.

By pressing the “Form” button we can switch back to the *form mode* and the XML document is read to fill the form with new data (see figure 5.5).

Schema: example1.xsd		Choose	Reload	Delete metadata
XSD	Form	XML	Metadata	Save
		Reset	<input checked="" type="checkbox"/> Auto save	

shiporder

* orderid: 889923

orderperson: John Smith

shipto

name: Ola Nordmann

address: Langgt 23

city: 4000 Stavanger

country: Norway

item

1

title: Empire Burlesque

note: Special Edition

quantity: 1

price: 10.90

Remove

2

title: Hide your heart

Add note

quantity: 1

price: 9.90

Remove

Add item

Figure 5.5. A sample form populated with the new XML data.

As we can see the user is able to change same data in two different modes – *form mode* and *XML mode*. This is possible as the generated web form is able to write and read XML instances. We could also use this functionality to invoke web services for example.

We have demonstrated how to change the data in Form mode as well as in XML mode. Further we will proceed with customizing the appearance of the *Form* itself by editing the DynaData.

5.4. Customization

DynaForm enables a form developer to customize the generated web form using a presentation specification language called **DynaData**. If omitted it is automatically generated based on the default values. In our example the DynaData mode is shown on figure 5.6.

```
@CUSTOMIZED

@BROKEN

@GENERATED
//shiporder { label: "shiporder"; section-style: inherit; }
//orderid { label: "orderid"; readonly: false; required: true; control: Input; }
//orderperson { label: "orderperson"; readonly: false; required: false; control: Input; }
//shipto { label: "shipto"; section-style: inherit; }
//name { label: "name"; readonly: false; required: false; control: Input; }
//address { label: "address"; readonly: false; required: false; control: Input; }
//city { label: "city"; readonly: false; required: false; control: Input; }
//country { label: "country"; readonly: false; required: false; control: Input; }
//item { label: "item"; section-style: inherit; repeat-style: normal; }
//title { label: "title"; readonly: false; required: false; control: Input; }
//note { label: "note"; readonly: false; required: false; control: Input; repeat-style: normal; }
//quantity { label: "quantity"; readonly: false; required: false; control: Input; }
//price { label: "price"; readonly: false; required: false; control: Input; }

/shiporder { label: "shiporder"; section-style: inherit; }
/shiporder/orderid { label: "orderid"; readonly: false; required: true; control: Input; }
/shiporder/orderperson { label: "orderperson"; readonly: false; required: false; control: Input; }
/shiporder/shipto { label: "shipto"; section-style: inherit; }
/shiporder/shipto/name { label: "name"; readonly: false; required: false; control: Input; }
/shiporder/shipto/address { label: "address"; readonly: false; required: false; control: Input; }
/shiporder/shipto/city { label: "city"; readonly: false; required: false; control: Input; }
/shiporder/shipto/country { label: "country"; readonly: false; required: false; control: Input; }
/shiporder/item { label: "item"; section-style: inherit; repeat-style: normal; }
/shiporder/item/title { label: "title"; readonly: false; required: false; control: Input; }
/shiporder/item/note { label: "note"; readonly: false; required: false; control: Input; repeat-style: normal; }
/shiporder/item/quantity { label: "quantity"; readonly: false; required: false; control: Input; }
/shiporder/item/price { label: "price"; readonly: false; required: false; control: Input; }
```

Figure 5.6. The initial DynaData of the sample form.

In short everything generated is placed after the `@GENERATED` marker. Unless we move some lines to the `@CUSTOMIZED` part they will be ignored and overwritten. The DynaData syntax was covered in chapter 4.5 on page 24.

Assume we have changed the DynaData as shown on figure 5.7.

Schema: **example1.xsd**
Choose
Reload
Delete metadata

XSD
Form
XML
Metadata
Save
Reset

```

@CUSTOMIZED
//shiporder { label: "Ship Order"; section-style: inherit; }
//orderid { label: "Order ID"; readonly: false; required: true; control: Input; }
//orderperson { label: "Order Person"; readonly: false; required: false; control: Input; }
//shipto { label: "Ship To"; section-style: inherit; }
//name { label: "Name"; readonly: false; required: false; control: Input; }
//address { label: "Address"; readonly: false; required: false; control: Input; }
//city { label: "City"; readonly: false; required: false; control: Input; }
//country { label: "Country"; readonly: false; required: false; control: Input; }
//item { label: "Items"; section-style: inherit; repeat-style: table; }
//title { label: "Title"; readonly: false; required: false; control: Text; size: 30; }
//note { label: "Note"; readonly: false; required: false; control: TextArea; repeat-style: normal; }
//quantity { label: "Quantity"; readonly: false; required: false; control: Text; size: 3; }
//price { label: "Price"; readonly: false; required: false; control: Text; size: 5; }

@BROKEN

@GENERATED

```

Figure 5.7. The customized DynaData of the sample form.

We moved all the lines starting with `//` to the `@CUSTOMIZED` part. We changed the labels to begin with an upper case letter. Instead the general *Input* control we assigned *Text* control with custom size attribute to some of the *Form Elements*. The “note” was set a *TextArea* control. The repeat style of “item” was set to *Table*.

Returning to the *form mode* we get the form with same data but a new look (see figure 5.8).

Schema: **example1.xsd**
Choose
Reload
Delete metadata

XSD
Form
XML
Metadata
Save
Reset
☒ Auto save

Ship Order

* Order ID:

Order Person:

Ship To

Name:

Address:

City:

Country:

Items

#	Title		Quantity	Price	
1	<input type="text" value="Empire Burlesque"/>	Note: <input type="text" value="Special Edition"/> <input type="button" value="Remove"/>	<input type="text" value="1"/>	<input type="text" value="10.90"/>	<input type="button" value="Remove"/>
2	<input type="text" value="Hide your heart"/>	<input type="button" value="Add Note"/>	<input type="text" value="1"/>	<input type="text" value="9.90"/>	<input type="button" value="Remove"/>

Figure 5.8. The customized sample form.

As we see the labels have changed and the “item”-s are now rendered as a table. Also “note” contains a large text box and “quantity” and “price” are a bit smaller.

As we have shown how to customize a form we will now change the XML schema itself to show how the evolution of the schema is supported in terms of customization.

5.5. Schema Changes

Suppose the schema changed a bit:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
            <xs:element name="zip" type="xs:integer"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="productName" type="xs:string"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

</xs:schema>
```

The “shipto” now contains additional “zip” element, “note” element was removed and “title” element was renamed as “productName”.

A new empty form with same DynaData is shown on figure 5.9:

Figure 5.9. The customized sample form.

The form is now generated based on the new schema. The “note” element is removed. The “zip” and “productName” have no custom labels as they are new. For everything else same DynaData was still used successfully.

The DynaData itself is automatically updated (see figure 5.10):

```

@CUSTOMIZED
//shiporder { label: "Ship Order"; section-style: inherit; }
//orderid { label: "Order ID"; readonly: false; required: true; control: Input; }
//orderperson { label: "Order Person"; readonly: false; required: false; control: Input; }
//shipto { label: "Ship To"; section-style: inherit; }
//name { label: "Name"; readonly: false; required: false; control: Input; }
//address { label: "Address"; readonly: false; required: false; control: Input; }
//city { label: "City"; readonly: false; required: false; control: Input; }
//country { label: "Country"; readonly: false; required: false; control: Input; }
//item { label: "Items"; section-style: inherit; repeat-style: table; }
//quantity { label: "Quantity"; readonly: false; required: false; control: Text; size: 3; }
//price { label: "Price"; readonly: false; required: false; control: Text; size: 5; }

@BROKEN
//title { label: "Title"; readonly: false; required: false; control: Text; size: 30; }
//note { label: "Note"; readonly: false; required: false; control: TextArea; repeat-style: normal; }

@GENERATED
//zip { label: "zip"; readonly: false; required: false; control: Input; }
//productName { label: "productName"; readonly: false; required: false; control: Input; }

//shiporder/shipto/zip { label: "zip"; readonly: false; required: false; control: Input; }
//shiporder/item/productName { label: "productName"; readonly: false; required: false; control: Input; }

```

Figure 5.10. The new DynaData of the sample form.

The “title” and “note” elements are now marked @BROKEN as they were not found in the form and new lines were generated in the @GENERATED part as the corresponding form elements were not included in the @CUSTOMIZED part.

We know that “title” was renamed as “productName” so we update the corresponding line in the @BROKEN part. We remove the “note” element, put the “zip” element to the @CUSTOMIZED part and change its label. After applying the changes new DynaData is shown on figure 5.11.

Schema: example1.xsd	Choose	Reload	Delete metadata
XSD	Form	XML	Metadata
Save	Reset		

```

@CUSTOMIZED
//shiporder { label: "Ship Order"; section-style: inherit; }
//orderid { label: "Order ID"; readonly: false; required: true; control: Input; }
//orderperson { label: "Order Person"; readonly: false; required: false; control: Input; }
//shipto { label: "Ship To"; section-style: inherit; }
//name { label: "Name"; readonly: false; required: false; control: Input; }
//address { label: "Address"; readonly: false; required: false; control: Input; }
//city { label: "City"; readonly: false; required: false; control: Input; }
//country { label: "Country"; readonly: false; required: false; control: Input; }
//item { label: "Items"; section-style: inherit; repeat-style: table; }
//quantity { label: "Quantity"; readonly: false; required: false; control: Text; size: 3; }
//price { label: "Price"; readonly: false; required: false; control: Text; size: 5; }
//productName { label: "Title"; readonly: false; required: false; control: Text; size: 30; }
//zip { label: "ZIP"; readonly: false; required: false; control: Input; }

@BROKEN

@GENERATED

```

Figure 5.11. The fixed DynaData of the sample form.

The @BROKEN part is now empty as we removed the “note” element and the “productName” (former “title”) element was automatically moved to the @CUSTOMIZED part. The latter now covers the whole form. Therefore the @GENERATED part is also empty.

If we press the “Form” button we see that the labels of the form are now updated (figure 5.12).

Schema: example1.xsd Choose Reload Delete metadata

XSD **Form** XML Metadata Save Reset ☒ Auto save

Ship Order

* Order ID:

Order Person:

Ship To

Name:

Address:

City:

Country:

ZIP:

Items

#	Title	Quantity	Price
1	<input type="text"/>	<input type="text"/>	<input type="text"/>

Add Items

Figure 5.12. The sample form with fixed customization.

So we demonstrated that if the schema changes new and (re)moved elements are automatically detected and added to corresponding parts of the DynaData. Therefore we see what data should be reviewed. Handling the schema changes is easy as we have to do manual work as little as possible – only changing particular labels, matching the old and new element names etc. We do not have to write the DynaData from scratch for new elements.

This finishes our walkthrough of DynaForm. We continue by summarizing the thesis.

Conclusions

The aim of this thesis was to design and implement a **web form generator** producing forms out of XML schemas and presentation specifications which are allowed to evolve independently. As the presentation specification we meant details like labels, user interface controls etc for any particular XML schema element.

We implemented the web form generator called **DynaForm**. Although this implementation takes the form of a reusable component built on top of the Aranea Web Framework, it is to a large extent framework-independent. The DynaForm component takes as input an XML Schema, an optional presentation specification written in a language called **DynaData**, and an optional XML instance. Based on these inputs, the component generates a web form, an updated version of the DynaData specification to reflect any changes observed in the schema, and an XML instance filled with the form's input data.

A DynaData specification is composed of independent elements each capturing an assertion on how a given XML schema element should be rendered. Each time the DynaData file is synchronized with the schema DynaForm identifies which schema elements are covered by the DynaData assertions, which schema elements are not covered by any assertion, and also which DynaData assertions refer to a non-existent schema element. According to this distinguishing the DynaData file is divided into **Custom**, **Generated** and **Broken** sections. The two latter sections describe the schema elements which were new or removed respectively.

By means of an end-to-end scenario, we showed how DynaForm generated a web form for a sample schema. The scenario illustrated how DynaData assertions are automatically generated (using default values) for each XML schema element not covered by any existing DynaData assertion. This step illustrated one of the key features of the proposed solution: If the Web form developer has written an assertion for rendering a particular type of element in a schema, that assertion is used during the form generation, but whenever an assertion is missing, a default value is used to render the corresponding element.

We also showed how to customize the form by altering a DynaData presentation specification and we illustrated how the solution works when the underlying schema

changes. The scenario allowed us to demonstrate that after a schema change, DynaForm is still able to continue producing a form under the new schema (in a degraded mode). The Web form developer can then check the changed parts of the DynaData specification to determine which additional assertions need to be added in the specification to fully adapt the form layout in response to the change. All in all, this approach provides an incremental and zero-downtime approach to maintaining Web forms up to date with respect to their underlying schema.

Altogether we demonstrated that a web form can be generated using a schema and a presentation specification which are allowed to evolve both independently. The current prototype implementation of DynaForm is built on top of the Aranea Web Framework. A possible direction for further work is to extend the DynaForm tool in order to add support for other Web frameworks, such as Spring MVC or Java Server Faces. Also, the current layout of complex XSD types in DynaData is rather simplistic. By implementing more sophisticated automated layout techniques we could probably improve the look-and-feel of the forms generated by DynaForm. Finally, the set of supported XSD elements in DynaForm is limited as discussed in Chapter 2. Lifting these restrictions would be a requirement to make DynaForm useful in practice.

Dünaamiline *schema*-põhine veebivormide genereerimine Javas

Rein Raudjärv

Magistritöö

Kokkuvõte

Käesoleva töö eesmärk on kavandada ja realiseerida **veebivormide generaator**, mille aluseks on XML *schema*-d ning esituskirjeldused, mis võivad kumbki iseseisvalt muutuda. Esituskirjelduse all peame silmas XML *schema* elementide üksikasju nagu sildid, kasutajaliidese komponendid jne.

Me realiseerime veebivormide generaatori nimega **DynaForm**. Tegemist on taaskasutatava veebikomponendiga, mis baseerub küll **Aranea veebiraamistikul**, kuid sellest suurem osa ei sõltu antud raamistikust. Antud komponendi sisenditeks on XML *schema*, **DynaData** keeles kirjutatud esituskirjeldus ning XML dokument. Nendest kohustuslik on ainult esimene. Komponendi väljunditeks on genereeritud veebivorm, genereeritud/uuendatud DynaData fail ning vormi andmetega täidetud XML dokument.

DynaData esituskirjeldus koosneb kirjetest, mis panevad paika kuidas mingi XML *schema* element peaks välja nägema. Iga kord kui DynaData faili XML *schema*-ga sünkroniseeritakse tehakse kindlaks, milliseid *schema* elemente DynaData kirjed käsitlesid, milliseid mitte ning missuguseid kirjeid ei saanud rakendada kuna vastavaid *schema* elemente ei leitud. Vastavalt sellele jagatakse DynaData fail **Custom**, **Generated** ning **Broken** nimelisteks lõikudest.

Me vaatleme stsenaariumit, kus DynaForm genereerib näidis XML *schema* jaoks veebivormi. Näeme kuidas XML *schema* elementide jaoks, mille kohta DynaData kirjed puuduvad koostatakse automaatselt vaikimisi esitustavade alusel uued DynaData kirjed. See samm illustreerib loodava lahenduse ühte põhitunnust: kui vormi arendaja on kirjeldanud mingi *schema* elemendi väljanägemist, siis vormi genereerimisel seda ka kasutatakse, kui aga vastav kirje puudub, siis kasutatakse elemendi kuvamisel vaikeväärtusi.

Samuti näitame kuidas DynaData esitluskirjest muutes saab vormi kohandada ning et antud lahendus töötab ka pärast seda, kui *schema* muutub – DynaForm suudab vormi genereerida ka uue *schema* põhjal (kehvemal kujul). Veebivormi arendaja saab sel juhul

DynaData faili muutunud kohad üle käia ja viia need vastavusse *schema* muutustega. Seega võimaldab antud lahendus veebivorme uuendada aluseks oleva *schema* suhtes järgult ning ilma katkestusteta.

Bibliography

- [Be03] H. Bergsten. *JavaServer pages (3rd edition)*. O'Reilly Media, Inc., 2003
- [Be04] H. Bergsten. *JavaServer faces*. O'Reilly Media, Inc., 2004
- [Ce02] E. Cerami. *Web services essentials*. O'Reilly Media, Inc., 2002
- [DK00] A. van Deursen, P. Klint, and J. Visser, *Domain-specific languages: An annotated bibliography*. SIGPLAN Notices, vol. 35, no. 6, pp. 26–36, 2000.
- [Du03] M. Dubinko. *XForms essentials*. O'Reilly Media, Inc., 2003
- [Fl06] D. Flanagan. *JavaScript: the definitive guide (5th edition)*. O'Reilly Media, Inc., 2006
- [FV04] M. Florins, J. Vanderdonckt. *Graceful degradation of user interfaces as a design method for multiplatform systems. Proceedings of the 2004 International Conference on Intelligent User Interfaces*, pages 140-147, ACM Madeira, Portugal, 2004.
- [GH94] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994
- [GJ00] J. Gosling, B. Joy, G. Steele, G. Bracha. *JavaTM language specification (3rd edition)*. Prentice Hall PTR, 2005. <http://java.sun.com/docs/books/jls/> (May 22 2010)
- [HC01] J. Hunter, W. Crawford. *Java servlet programming (2nd edition)*. O'Reilly Media, Inc., 2001
- [HM04] E. R. Harold, W. S. Means. *XML in a nutshell (3rd edition)*. O'Reilly Media, Inc., 2004
- [Ho08] A. T. Holdener. *AJAX: the definitive guide*. O'Reilly Media, Inc., 2008
- [Ka10] J. Kabanov. *Aranea Introductory Tutorial*. 2010
<http://www.araneaframework.org/docs/intro/html/> (May 22 2010)
- [Kw10] K. Kawaguchi. *XSOM User's Guide*. 2010
<https://xsom.dev.java.net/userguide.html> (May 22 2010)
- [LD06] S. Ladd, D. Davison, S. Devijver, C. Yates. *Expert Spring MVC and Web Flow*. Apress, 2006

- [LT06] R. Lerdorf, K. Tatroe, P. MacIntyre. *Programming PHP (2nd edition)*. O'Reilly Media, Inc., 2006
- [Me06] E. A. Meyer. *CSS: the definitive guide*. O'Reilly Media, Inc., 2006
- [MK96] C. Musciano, B. Kennedy. *HTML: the definitive guide*. O'Reilly & Associates, 1996
- [MK06] O. Mürk, J. Kabanov. *Aranea: web framework construction and integration kit*. In *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, 163-172, ACM Press New York, NY, USA, 2006.
- [SB08] B. Smeets, U. Boness, R. Bankras. *Beginning Google Web Toolkit: From Novice to Professional*. Apress, 2008
- [Ra03] E. T. Ray. *Learning XML*. O'Reilly Media, Inc., 2003
- [Ti01] D. Tidwell. *XSLT*. O'Reilly Media, Inc., 2001
- [Wa02] P. Walmsley. *Definitive XML schema*. Prentice. Hall PTR, 2002

Appendix 1

Implementation Details

In this appendix we give an overview of the DynaForm implementation in Java by introducing the main interfaces and some of the simpler classes.

1. Overview

In the requirements section we saw the DynaForm as a single machine with certain inputs and outputs. In the implementation level different tasks such as XML operations, DynaData operations and rendering are assigned to separate components.

In the actual implementation the **Form Builder** takes an **XML schema** as a single input and produces three outputs: a **Form**, an **XML Reader** and an **XML Writer** (see figure 1). The generated **Form** stores the form data and provides methods for reading and updating current values. It also keeps a list of listeners receiving data update events. This allows other components to read and modify the **Form** as well as be sync with it. The generated **XML Reader** enables to read an XML instance and update the **Form**. The **XML Writer** enables to observe the **Form** and write an XML instance accordingly.

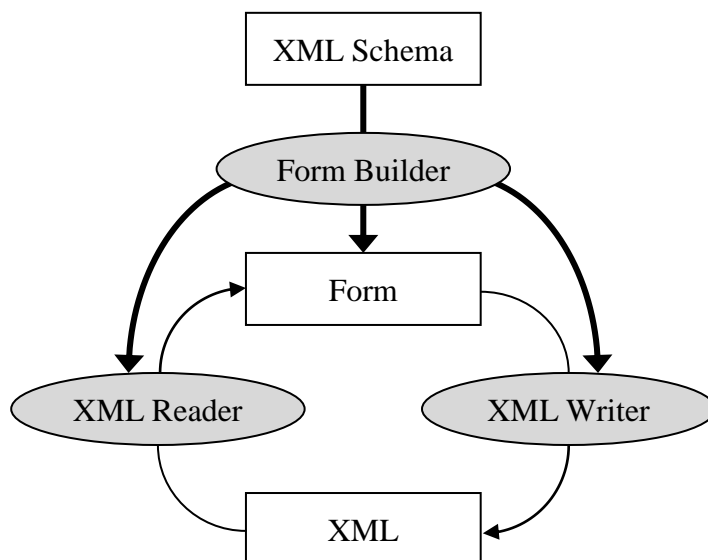


Figure 1. The inputs and outputs of Form Builder.

By distributing different tasks the *Form* is released from the following aspects:

1. **Independence of XML** – the generated *Form* is unable to read or write XML by itself. Instead the generated *XML Reader* and *XML Writer* are capable for this by observing or updating the corresponding *Form* accordingly.
2. **Independence of web frameworks** – the generated *Form* cannot represent itself to a user. To render the data the *Form* must be converted into component(s) of a particular web framework. The implementation contains the *Aranea Form Builder* which takes a *Form* as an input and produces a hierarchy of Aranea Widgets as output. The builder links the generated *Widgets* with the *Form* by automatically keeping them in sync.
3. **Independence of DynaData** – the generated *Form* stores the data in memory but it is not capable of reading or writing a DynaData file. Instead a **DynaData** instance is created by providing it with a *Form*. It always first scans the *Form* to read the default values. After that the *DynaForm* instance is able to read or write a DynaData file at any time.

In the following chapters we observe the *Form* and each of these aspects in detail. The final two chapters will show how to embed the generated *Form* into an Aranea Widget and how to access the DynaData API.

1. Form

A *Form* stores the form data and provides methods for reading and updating current values. It also keeps a list of listeners receiving data update events.

In the architecture section we described the form also as a **hierarchical composition** and gave a set of component types – *Form Element*, *Form Section*, *Form Repeat*, *Form Sequence*, *Form Choice* and *Form All*. This means that any of these components is a *Form* and except for *Form Element* can contain other *Forms*.

In Java these component types are interfaces which all extend the same parent:

```
interface Form { ... }

interface FormElement extends Form { ... }
interface FormSection extends Form { ... }
interface FormRepeat extends Form { ... }
interface FormSequence extends Form { ... }
interface FormChoice extends Form { ... }
interface FormAll extends Form { ... }
```

Although in the architecture we connected each component type with a visual example the *Form* components cannot render themselves. The examples represented their counterparts produced by the *Aranea Form Builder*.

We continue by describing each of the *Form* interfaces.

1.1. Form Element

To recall a *Form Element* represents a schema *<element>* of simple type or an *<attribute>*. It is a labeled single form field displaying a current value and allowing user to alter it.

The `FormElement` interface contains the following methods:

```
interface FormElement<E> extends Form {

    Data<E> getData();

    Restrictions<E> getRestrictions();

    String getLabel();
    void setLabel(String label);

    Control getControl();
    void setControl(Control control);

    boolean isReadOnly();
    void setReadOnly(boolean readOnly);

    boolean isRequired();
    void setRequired(boolean required);

    boolean isDisabled();
    void setDisabled(boolean disabled);

    // Listeners

    void addListener(ElementChangeListener<E> listener);
    void removeListener(ElementChangeListener<E> listener);

    interface ElementChangeListener<E> {
        void onSetValue(FormElement<E> element);
        void onSetLabel(FormElement<E> element);
        void onSetControl(FormElement<E> element);
        void onSetReadOnly(FormElement<E> element);
        void onSetRequired(FormElement<E> element);
        void onSetDisabled(FormElement<E> element);
    }
}
```

We see a number of getter and setter methods for properties we already listed in the architecture section. In addition `ElementChangeListener` instances can be registered that will receive a corresponding event after each setter is called.

The actual field value is contained in a `Data` instance and the restrictions in the `Restrictions` object. Both of them are parameterized with the field type (parameter `E`).

We continue by inspecting `Data`, `Restrictions` and `Control` types.

1.1.1. Data

The *Form Builder* associates each *Form Element* with a particular XML simple type. On the other hand we want to use Java types for the form field values. Moreover same Java type (e.g. `java.util.Date`) may correspond to many XML types with certain distinctions.

The `Data` abstraction enables to see the same value as an XML type as well as a Java type and provide custom rules for the conversion. The `Data` interface has the following methods:

```
interface Data<E> {
    Class<E> getType();
    E getValue();
    void setValue(E value);
    String getXmlValue();
    void setXmlValue(String value);
}
```

The parameter `E` refers to the Java type of the value. The corresponding class is returned by the `getType()` method.

As we can see `getValue()` and `setValue()` are used to get/set a value of Java type. At the same `getXmlValue()` and `setXmlValue()` enable to get/set a value of XML type.

The *Form Builder* assigns each `FormElement` an appropriate `Data` instance based on the XML simple type. If no match is found a base type is being used to find the match. E.g. `positiveInteger` has no match but the base type `integer` does.

The supported XML simple types with matching Java types wrapped by `Data` objects are listed in Table 1.

XML simple type	Java type	Pattern
string	String	
integer	BigInteger	
long	Long	
int	Integer	
short	Short	

byte	Byte	
decimal	BigDecimal	
date	java.util.Date	yyyy-MM-dd
time	java.util.Date	HH:mm:ss
datetime	java.util.Date	yyyy-MM-dd'T'HH:mm:ss
boolean	Boolean	

Table 1. Supported XML simple types with corresponding Java types wrapped by Data objects.

1.1.2. Restrictions

Each `FormElement` contains `Restrictions` which correspond to the restrictions assigned to the XML simple type. DynaForm currently supports **enumeration**, **length** and **number range** restrictions:

```
interface Restrictions<E> {

    // Enumeration
    List<Choice<E>> getChoices();
    public interface Choice<E> {
        String getLabel();
        void setLabel(String label);
        E getValue();
        void setValue(E value);
    }

    // Length
    Integer getLength();
    void setLength(Integer length);
    Integer getMinLength();
    void setMinLength(Integer minLength);
    Integer getMaxLength();
    void setMaxLength(Integer maxLength);

    // Number Range

    E getMinInclusive();
    void setMinInclusive(E minInclusive);
    E getMinExclusive();
    void setMinExclusive(E minExclusive);
    E getMaxInclusive();
    void setMaxInclusive(E maxInclusive);
    E getMaxExclusive();
    void setMaxExclusive(E maxExclusive);
}
```

Enumeration choices and ranges are parameterized with field type (parameter `E`) as they contain same values that can be assigned to the field. Length conditions are always integers as they refer to length not the field value itself.

The same properties were already listed in the architecture section and they meaning is obvious.

1.1.3. Controls

Each *Form Element* has a *Control* – a user interface component that displays and allows changing the current value. Although it seems we are already dealing with the web layer it is not that. In this context by a *Control* we only mean its name plus optional parameters for the size etc.

The actual web control is created when the web layer is created (e.g. by the *Aranea Form Builder*). To be able to extend the set *Controls* they form a hierarchy – e.g. `Secret` extends `Text`. This enables the actual web layer to render the parent control type if the sub type is not supported. On the other hand if a more general *Control* is used the actual web layer may choose which implementation to choose (e.g. whether a `SelectOne` should be displayed as a drop-down or a list of radio buttons).

All *Control* types inherit the `Control` interface (see architecture section for the descriptions):

```
interface Control {

    /** A general input control */
    interface Input extends Control {}

    /** A single line text input. */
    interface Text extends Input {
        // Specifies the width (in characters)
        Integer getSize();
        void setSize(Integer size);
        // Specifies the maximum length (in characters)
        Integer getMaxLength();
        void setMaxLength(Integer maxLength);
    }

    /** Like "text", but the input characters are hidden. */
    interface Secret extends Text {}

    /** A multi-line text input. */
    interface TextArea extends Control {
        // Specifies the visible width
        Integer getCols();
        void setCols(Integer cols);
        // Specifies the visible number of rows
        Integer getRows();
        void setRows(Integer rows);
    }

    /** Renders a value but provides no means for changing data. */
    interface Output extends Control {}

    /** Allows the user to make a single selection from a set of choices.
    */
    interface SelectOne extends Control {}
}
```

```

interface SelectBox extends SelectOne {
    // Specifies the visible number of rows
    Integer getSize();
    void setSize(Integer size);
}

interface ComboBox extends SelectOne {}

interface RadioSelectBox extends SelectOne {}

```

As we see `Input`, `TextArea`, `Output` and `SelectOne` extend the `Control` interface directly.

`Text` extends the `Input` by adding `size` and `maxLength` properties. `Secret` extends the latter by hiding the characters entered. `TextArea` does not extend `Text` because of the differences in the set of properties.

`SelectOne` has three sub-interfaces: `SelectBox` displays a box with the given size, `ComboBox` enables to choose an item from the drop-down menu and `RadioSelectBox` displays a radio button for each item.

As we finished with `FormElement` we continue with the rest of the `Form` interfaces.

1.2. Form Section

A *Form Section* represents a schema <element> of complex type. It always contains another *Form* as the content plus a label.

The `FormSection` interface has the following methods:

```

interface FormSection<F extends Form> extends Form {

    F getContent();

    String getLabel();
    void setLabel(String label);

    enum SectionStyle { HORIZONTAL, VERTICAL, ROW, COLUMN, INHERIT }
    SectionStyle getSectionStyle();
    void setSectionStyle(SectionStyle sectionStyle);

    // Listeners

    void addListener(SectionChangeListener<F> listener);
    void removeListener(SectionChangeListener<F> listener);

    interface SectionChangeListener<F extends Form> {
        void onSetLabel(FormSection<F> section);
        void onSetSectionStyle(FormSection<F> section);
    }
}

```

The contained sub-component is returned by `getContent()` method. The `FormSection` is also parameterized with the sub-component type (parameter `F`) which must extend the `Form` interface.

There are also getter and setter methods for two properties – label and *Section Style*. The latter is one of the enumeration values (see chapter 4.4.1 page 21 for the descriptions). In addition `SectionChangeListener` instances can be registered that will receive a corresponding event after each setter is called.

1.3. Form Repeat

A *Form Repeat* represents a repetition of schema element (`maxOccurs > 1`). It is also used for optional elements (`minOccurs = 0` and `maxOccurs = 1`). A *Form Repeat* contains a dynamic number of instances of same sub-component.

The `FormRepeat` interface has the following methods:

```
interface FormRepeat<F extends Form> extends Form {

    Integer getMin();
    Integer getMax();

    List<F> getChildren();

    F add();
    void remove(F child);
    void clear();

    enum RepeatStyle { NORMAL, ROWS, COLUMNS, TABLE }
    RepeatStyle getRepeatStyle();
    void setRepeatStyle(RepeatStyle repeatStyle);

    // Listeners

    void addListener(RepeatChangeListener<F> listener);
    void removeListener(RepeatChangeListener<F> listener);

    interface RepeatChangeListener<F extends Form> {
        void onAdd(F child);
        void onRemove(F child);
        void onSetRepeatStyle(FormRepeat<F> sequence);
    }
}
```

`getMin()` and `getMax()` return the corresponding limits to the number of children.

`getChildren()` returns the current list of sub-components. The `FormRepeat` is also parameterized with the sub-component type (parameter `F`) which must extend `Form`.

Notice that `add()` method has no arguments and it somehow returns a new child. When `FormRepeat` is created it is provided with a `Factory`:

```
interface Factory<E> { E create(); }
```

This enables `FormRepeat` to create new children.

The `remove()` method takes an existing child and removes it from the list. `clear()` removes all children.

There are also getter and setter method for *Repeat Style*. The latter is one of the enumeration values (see chapter 4.4.2 on page 23 for the descriptions).

In addition `RepeatChangeListener` instances can be registered that will receive corresponding events after a setter is called, a new child is added or an existing is removed.

1.4. Form Sequence

A *Form Sequence* represents a schema `<sequence>`. It is also used to group together XML attributes and an XML element body. It contains a fixed list of sub-components each of which may be of different type.

The `FormSequence` interface has the following method:

```
interface FormSequence extends Form {  
  
    List<Form> getChildren();  
  
}
```

This is the simplest *Form* component type. `getChildren()` returns the fixed list of sub-components. The `FormSequence` is not parameterized as each of the children may be of different type. There are also no listeners as nothing can be changed.

1.5. Form Choice

A *Form Choice* represents a schema `<choice>`. It always contains a fixed list of sub-components each of which may be of different type. In XML exactly one of the sub-components must occur. So in the *Form* user must be able to select this component.

The `FormChoice` interface has the following methods:

```
interface FormChoice extends Form {  
  
    List<Form> getChildren();  
    Form getSelectedChild();  
    int getSelectedIndex();  
    void setSelectedIndex(int index);
```

```

// Listeners

void addListener(ChoiceChangeListener listener);
void removeListener(ChoiceChangeListener listener);

interface ChoiceChangeListener {
    void onSetSelectedIndex(int index);
}
}

```

`getChildren()` returns the fixed list of sub-components.

`getSelectedChild()` and `getSelectedIndex()` return the sub-component and its index currently selected. `setSelectedIndex()` enables to select a new sub-component.

In addition `ChoiceChangeListener` instances can be registered that will receive an event after new child is selected.

1.6. Form All

A *Form All* represents a schema `<all>`. It always contains a fixed list of sub-components each of which may be of different type. In XML any of the sub-components may occur but not more than once. The sub-components may occur in any order. So in the *Form* user must be able to enable or disable each component as well as choose their order.

The `FormAll` interface has the following methods:

```

interface FormAll extends Form {

    List<Form> getChildren();
    int[] getIndexes();
    boolean areAllChildrenRequired();

    List<Form> getSelectedChildren();
    int[] getSelectedIndexes();
    void setSelectedIndexes(int[] indexes);
    List<Form> getUnselectedChildren();
    int[] getUnselectedIndexes();

    // Listeners

    void addListener(AllChangeListener listener);
    void removeListener(AllChangeListener listener);

    interface AllChangeListener {
        void onSetSelectedIndexes(int[] indexes);
    }
}

```

`getChildren()` returns the fixed list of sub-components and `getIndexes()` gives their indexes (0, 1, ..., n-1).

`areAllChildrenRequired()` returns `true` if all sub-components must occur exactly once giving only an opportunity to reorder them. Otherwise each child may occur but not more than once.

`getSelectedChildren()` and `getSelectedIndexes()` return the sub-components and their indexes currently selected. `setSelectedIndexes()` enables to reselect sub-components including their order.

`getUnselectedChildren()` and `getUnselectedIndexes()` return the sub-components and their indexes currently not selected.

In addition `AllChangeListener` instances can be registered that will receive an event after new children are selected.

2. Writing and Reading XML

As we said in the overview of the implementation the *Form Builder* takes an **XML schema** as a single input and produces three outputs: a *Form*, an *XML Reader* and an *XML Writer*. The generated *Form* is unable to read or write XML by itself. Instead the generated *XML Reader* and *XML Writer* are capable for this by observing or updating the corresponding *Form* accordingly.

The main function of the *Form Builder* is the following:

```
class XmlFormBuilder {  
  
    static XmlForm build(XSSchemaSet set) { ... }  
  
    interface XmlForm {  
        Form getForm();  
        XmlWriter getWriter();  
        XmlReader getReader();  
    }  
}
```

The `build()` method actually takes a schema set – this is a DOM representation of one or more schemas parsed by the XML Schema Object Model (XSOM) library [Kw10] – as input and outputs an `XmlForm` which holds a *Form*, an `XmlWriter`, and an `XmlReader`.

Internally `XmlFormBuilder` uses a visitor pattern [GH94] and traverses the whole schema hierarchy by producing an `XmlForm` per each schema element. E.g. for a schema `<element>` of complex type with no attributes, first the child `XmlForm` is received by visiting the body, then new `XmlForm` is created containing a `FormSection`, an

`XmlElementWriter` and an `XmlElementReader` each of which wrap the corresponding components of the child `XmlForm`. This means that not only the *Form* is a hierarchical composition but also the `XmlWriter` and `XmlReader` form a hierarchical structure.

We continue by describing the `XmlWriter` and `XmlReader` interfaces in detail but first we examine the `XmlVisitor` which is used by both of them.

2.1. XML Event Handler

As we stated `XmlWriter` and `XmlReader` are hierarchical compositions. Therefore they have to produce or consume XML also by traversing the XML structure hierarchically enabling sub-components to process the sub-structure of XML. We cannot read an XML using the standard Simple API for XML (SAX) [HM04] handler because it processes the whole XML as a flat document.

The standard SAX handler in Java has the following methods:

```
interface ContentHandler {

    void startDocument();
    void endDocument();

    void startElement(String uri, String localName, String qName,
                     Attributes atts);
    void endElement(String uri, String localName, String qName);

    void characters(char ch[], int start, int length);
    ...
}
```

Same handler methods are invoked regardless the position in the XML. In a hierarchical XML parsing the handler must be able to process the events as well as pass the processing to another handler. For this we use the following hierarchical XML event handler:

```
interface XmlVisitor {

    XmlVisitor text(String s);
    ElementVisitor visitElement(String element);
    ElementVisitor visitEndBody();

    interface ElementVisitor {

        ElementVisitor visitAttribute(String name, String value);
        XmlVisitor visitBody();
        XmlVisitor visitEndElement();
    }
}
```


Each method handles the event as well as returns the next handler. It may also throw an `InvalidXmlException` if the given event is not supported (unsupported element or the maximum number of this element is already reached etc):

`text()` handles plain text.

`visitElement()` handles the start of an XML element and returns the `ElementVisitor` for it.

`visitAttribute()` handles an attribute.

`visitBody()` handles the start of the body and returns the `XmlVisitor` for it.

`visitEndBody()` handles the end of a body and returns the `ElementVisitor` for the XML element containing the body.

`visitEndElement()` handles the end of an element.

Comparing to SAX handler the `XmlVisitor` has separate methods for attributes as well as starting and ending a body. In case of SAX handler it is hard to distinguish an empty body with no body at all as same methods are still invoked. In case of `XmlVisitor` the starting and ending a body is made explicit.

The `XmlVisitor` is swapped to an `ElementVisitor` for a temporary state – “we are inside the element but not inside the body”.

We continue by examining how the `XmlVisitor` is used for writing and reading XML.

2.2. XML Writer

The *XML Writer* is used to observe the generated *Form* and produce a corresponding XML instance.

It could directly output a string but in some case the output is required to be compact, in other case sub elements should be intended or the whole document must be wrapped to a certain width in characters etc. In short it is wise to make the `XmlWriter` just produce XML events and use a suitable handler to actually construct the XML string. For the event handling we chose to stick to the `XmlVisitor` although for writing we could also use the standard SAX handler.

The `XmlWriter` has two methods:

```
interface XmlWriter {  
  
    boolean isEmpty();  
    void write(XmlVisitor xv);  
  
}
```

`isEmpty()` returns `true` if there is anything to write. This is used by parent writes so they know whether to start the body at all. It is usually `false` if the form field is left blank.

`write()` takes the `XmlVisitor` and produces the XML events necessary. It also passes the process to sub-writers by giving them the corresponding `XmlVisitor` and `ElementVisitor` instances returned by the event handling methods.

2.3. XML Reader

The *XML Reader* is used to read an XML and update the generated *Form* accordingly.

Here we have to use `XmlVisitor` as the XML event handler as each *XML Reader* is only able to handle certain part of the whole document. For the sub-components it passes the handling to the sub-readers. Also when the certain part is handled it passes the reading to the next handler. So as soon as we know the next reader we can construct an `XmlVisitor` that will handle the certain part of the document, optionally passing the processing to its children and finally returning the next reader that was first assigned.

The `XmlReader` has only one method:

```
interface XmlReader {  
  
    XmlVisitor create(XmlReader next);  
  
}
```

The `create()` method returns the `XmlVisitor` for handling a particular part of the XML document. Last method that handles a supported event returns the next handler. The latter is created by invoking `next.create(null)`. `null` is also passed if the root *XML Reader* is invoked. If the XML document to be read is correct these `null`-s are actually never invoked.

We continue by describing some of the `XmlWriter` and `XmlReader` implementations.

2.4. Form Element Writer

As *Form Elements* may represent XML elements as well as XML attributes we introduce another abstraction for providing any text-based value.

The `TextWriter` has only one method:

```
interface TextWriter { String getText(); }
```

The `FormElementWriter` implements `TextWriter` instead of `XmlWriter`:

```
public class FormElementWriter implements TextWriter {
    private FormElement element;
    FormElementWriter(FormElement element) {
        this.element = element;
    }
    String getText() {
        return element.getXmlValue();
    }
}
```

The `getText()` method just invokes the `getXmlValue()` of the `FormElement`.

Defining a `TextWriter` instead of `XmlWriter` enables to use the same implementation both for an attribute as well as plain XML content. In the first case we only allow using a `TextWriter` as the attribute cannot be written by an `XmlWriter` which may e.g. produce another XML element into an attribute value. In case we need to write the `FormElement` into a body of an XML element we use the adapter.

`TextXmlWriter` wraps any `TextWriter` into an `XmlWriter`:

```
class TextXmlWriter implements XmlWriter {
    TextWriter textWriter;
    TextXmlWriter(TextWriter textWriter) {
        this.textWriter = textWriter;
    }
    TextWriter getWriter() { return textWriter; }
    boolean isEmpty() { return textWriter.getText() == null; }
    void write(XmlVisitor xv) { xv.text(textWriter.getText()); }
}
```

So to get an `XmlWriter` for a `FormElement` the adapter must be used followingly:

```
XmlWriter xw = new TextXmlWriter(new FormElementWriter(formElement));
```

2.5. Form Element Reader

As *Form Elements* may represent XML elements as well as XML attributes we also introduce abstraction for reading any text-based value.

The `TextHandler` has only one method:

```
interface TextHandler { void text(String s); }
```

The `FormElementReader` implements `TextHandler` instead of `XmlReader`:

```
class FormElementHandler implements TextHandler {
    FormElement element;
    FormElementHandler(FormElement element) {
        this.element = element;
    }
    void text(String s) { element.setXmlValue(s); }
}
```

In case of reading XML attributes we can use the `FormElementReader` directly. Otherwise we wrap it into a `TextXmlReader` which only accepts text in XML. Otherwise an `InvalidXmlException` is thrown.

`TextXmlReader` wraps any `TextHandler` into an `XmlReader`:

```
class TextXmlReader implements XmlReader {
    TextHandler handler;
    TextXmlReader(TextHandler handler) { this.handler = handler; }

    XmlVisitor create(XmlReader next) { return new TextVisitor(next); }
    class TextVisitor extends BaseXmlVisitor {
        TextVisitor(XmlReader next) { super(next); }
        XmlVisitor text(String s) {
            handler.text(s);
            return nextVisitor();
        }
    }
}
```

The `text()` method of `XmlVisitor` is just passed to corresponding `TextHandler`.

The `BaseXmlVisitor` rejects any events by default. It has the following methods:

```
class BaseXmlVisitor implements XmlVisitor {
    XmlReader nextReader;
    BaseXmlVisitor(XmlReader next) {
        this.nextReader = next;
    }
    XmlReader nextReader() { return nextReader; }
    XmlVisitor nextVisitor() {
        return nextReader == null ? null : nextReader.create(null);
    }
    XmlVisitor text(String s) {
        throw new InvalidXmlException();
    }
    ElementVisitor visitElement(String element) {
        throw new InvalidXmlException();
    }
    ElementVisitor visitEndBody() {
        throw new InvalidXmlException();
    }
}
```

The `nextVisitor()` method should be called when the current visitor is finished. This is exactly what `TextXmlReader` did after the `text()` method was handled.

So to get an `XmlReader` for a `FormElement` the adapter must be used followingly:

```
XmlReader xr = new TextXmlReader(new FormElementHandler(formElement));
```

2.6. XML Element Writer

The `XmlElementWriter` is used to write an XML element. It is created at the same time as the `FormSection` but they do not point to each other.

The `XmlElementWriter` has the following methods:

```
class XmlElementWriter implements XmlWriter {
    String name;
    Map<String, TextWriter> attributes;
    XmlWriter body;

    XmlElementWriter(String n, Map<String, TextWriter> as, XmlWriter body)
    {
        this.name = n; this.attributes = as; this.body = body;
    }

    boolean isEmpty() { return false; }

    void write(XmlVisitor xa) {
        ElementVisitor elementVisitor = xa.visitElement(name);
        if (attributes != null) {
            for (Entry<String, TextWriter> entry : attributes.entrySet()) {
                String attrName = entry.getKey();
                TextWriter textWriter = entry.getValue();
                String value = textWriter.getText();
                if (value != null)
                    elementVisitor.visitAttribute(attrName, value);
            }
        }
        if (body != null && !body.isEmpty()) {
            XmlVisitor bodyVisitor = elementVisitor.visitBody();
            body.write(bodyVisitor);
            bodyVisitor.visitEndBody();
        }
        elementVisitor.visitEndElement();
    }
}
```

The logic of `write()` is straightforward. We start the XML element, iterate through the `TextWriter`-s of the attributes, let the `XmlWriter` of the body do its job and finally finish the XML element.

As you can see `TextWriter` is used instead of `XmlWriter` for an attribute.

2.7. XML Element Reader

The `XmlElementReader` is used to read an XML element. It is created at the same time as the `FormSection` but they do not point to each other.

The XmlElementReader has the following methods:

```
class XmlElementReader implements XmlReader {
    String elemenetName;
    Map<String, TextHandler> attributes;
    XmlReader body;

    XmlElementReader(String n, Map<String, TextHandler> as, XmlReader body)
    {
        this.elemenetName = n; this.attributes = as; this.body = body;
    }
    public XmlVisitor create(XmlReader next) {
        return new XmlElementVisitor(next);
    }

    class XmlElementVisitor extends BaseXmlVisitor implements
    ElementVisitor {

        XmlVisitor bodyVisitor;

        XmlElementVisitor(XmlReader next) { super(next); }

        ElementVisitor visitElement(String element) {
            if (!elemenetName.equals(element))
                throw new InvalidXmlException();

            if (body != null)
                bodyVisitor = body.create(new EndBodyReader(this));
            return this;
        }

        ElementVisitor visitAttribute(String name, String value) {
            TextHandler handler = attributes == null ? null :
                attributes.get(name);

            if (handler == null)
                throw new InvalidXmlException();
            handler.text(value);
            return this;
        }

        public XmlVisitor visitBody() {
            if (body == null)
                throw new InvalidXmlException();
            return bodyVisitor;
        }

        public XmlVisitor visitEndElement() { return nextVisitor(); }
    }
}
```

The name of the element, the readers of the attributes and body are all provided to the XmlElementReader not the XmlElementVisitor. The latter is only enhanced with the next reader.

The XmlElementVisitor also implements the ElementVisitor as it is returned by the visitElement() and visitAttribute() methods.

`visitElement()` method checks the element name, prepares the body visitor and returns itself as an `ElementVisitor`. The body visitor is created from the body reader by attaching an `EndBodyReader`. The latter enables to end the body and return the current visitor.

`visitAttribute()` passes the value to the corresponding attribute handler and returns itself. An exception is thrown in case of unknown attribute.

`visitBody()` method returns the body visitor created. An exception is thrown in case no body visitor is assigned.

`visitEndElement()` returns the next visitor assigned to the `XmlElementVisitor`.

The `EndBodyReader` has the following methods:

```
class EndBodyReader implements XmlReader {
    ElementVisitor visitor;
    EndBodyReader(ElementVisitor elementVisitor) {
        this.visitor = elementVisitor;
    }
    XmlVisitor create(XmlReader next) {
        return new EndBodyVisitor(next);
    }
    class EndBodyVisitor extends BaseXmlVisitor {
        EndBodyVisitor(XmlReader next) { super(next); }
        ElementVisitor visitEndBody() { return visitor; }
    }
}
```

The `visitEndBody()` just returns the `ElementVisitor` provided.

2.8. Multi Writer

The `MultiWriter` is the plain writer composition. It is created at the same time as the `FormSequence` but they do not point to each other.

The `MultiWriter` has the following methods:

```
class MultiWriter implements XmlWriter {
    Collection<XmlWriter> children;
    MultiWriter(Collection<XmlWriter> children) { this.children = children; }
    boolean isEmpty() { return false; }
    void write(XmlVisitor xa) {
        for (XmlWriter child : children) child.write(xa);
    }
}
```

The `write()` methods just propagates the invocation to all of its children.

2.9. Multi Reader

The `MultiReader` is the plain reader composition. It is created at the same time as the `FormSequence` but they do not point to each other.

The `MultiReader` has the following methods:

```
class MultiReader implements XmlReader {

    static XmlReader newInstance(Collection<XmlReader> children) {
        if (children == null || children.isEmpty())
            return null;
        if (children.size() == 1)
            return children.iterator().next();
        return new MultiReader(items);
    }

    List<XmlReader> children;

    MultiReader(List<XmlReader> children) { this.children = children; }

    XmlVisitor create(XmlReader next) {
        XmlReader head = children.get(0);
        List<XmlReader> tail = children.subList(1, children.size());
        if (next != null) {
            tail = new ArrayList<XmlReader>(tail);
            tail.add(next);
        }
        return head.create(newInstance(tail));
    }
}
```

The `newInstance()` just checks the trivial cases when 0 or 1 children are passed.

The `create()` method delegates the invocation to the first child providing it with the rest of the children as the next `XmlReader`. If a next `XmlReader` was provided to the `MultiReader` itself it will be also appended (new `ArrayList` is created because the `subList()` only returns a view).

2.10. Writing and Reading Form Repeat, Choice and All

Writing and reading XML for the rest of the *Form* components depends on the state. E.g. *Form Repeat* contains dynamic number of elements and *Form Choice* and *Form All* have one or more elements selected. Therefore all of these writers and readers register themselves as listeners. We do not cover these implementations here as they are too complex. However the implementations can be found from the source code.

3. Rendering Form with Aranea

The generated *Form* cannot represent itself to a user. To render the data the *Form* must be converted into component(s) of a particular web framework. The implementation contains the *Aranea Form Builder* which takes a *Form* as an input and produces a hierarchy of **Aranea Widgets** as output. The builder links the generated *Widgets* with the *Form* by automatically keeping them in sync. Fortunately the builder must only convert the *Form* forgetting the *XML Reader* and *XML Writer*.

3.1. Form Visitors

To traverse the *Form* hierarchically we use the visitor pattern. The *Form* accepts two different visitors. One is a visitor that returns `void`, and the other is a “functor” that returns `Object`. The actual return type is parameterized by the corresponding visitor type.

The two visitors have the following methods:

```
interface FormVisitor {
    <E> void element(FormElement<E> element);
    <F extends Form> void section(FormSection<F> section);
    <F extends Form> void repeat(FormRepeat<F> sequence);
    void sequence(FormSequence composite);
    void choice(FormChoice choice);
    void all(FormAll all);
}

interface FormFunction<T> {
    <E> T element(FormElement<E> element);
    <F extends Form> T section(FormSection<F> section);
    <F extends Form> T repeat(FormRepeat<F> sequence);
    T sequence(FormSequence composite);
    T choice(FormChoice choice);
    T all(FormAll all);
}

interface Form {
    ...
    void accept(FormVisitor visitor);
    <T> T apply(FormFunction<T> function);
}
```

Any *Form* component accepts (applies) both visitors and both visitors can visit all *Form* components. In general a visitor first traverses the children and then combines the result with the current *Form* component.

The `FormVisitor` always return `void` and the `FormFunction` returns an instance of type parameter `T`.

3.2. Aranea Form Builder

The *Aranea Form Builder* produces an Aranea Widget per each *Form* component:

```
class AraneaFormBuilder implements FormFunction<UiForm> { ... }

interface UiForm extends Widget {
    boolean saveAndValidate();
}
```

The only method `saveAndValidate()` validates the data entered by user (returns true if success and false otherwise) and updates the *Form*. This command can be easily propagated through the Widget hierarchy as all children are required to implement the `UiForm` interface.

Although events like “Add” and “Remove” effect the *Form* immediately we have left the “Save” action to be explicit. Therefore *Form Elements* are updated only when the save action is called. In the future we could also use AJAX to save and validate the data on the fly.

The entire `AraneaFormBuilder` is long and complex so we demonstrate it only in terms of one *Form* component. We continue by observing how a *Form Choice* is transformed into an Aranea Widget.

3.3. Converting Form Choice

To demonstrate the event handling mechanism between the *Form* and Aranea Widgets we inspect how a *Form Choice* is transformed into an Aranea Widget.

The *Form Choice* visiting method of `AraneaFormBuilder` is the following:

```
class AraneaFormBuilder implements FormFunction<UiForm> {
    ...
    UiForm choice(FormChoice choice) {
        ChoiceFormWidget widget = new ChoiceFormWidget();

        // Send events from Widget to Form
        widget.setListener(new ChoiceEventListener() {
            void selectIndex(int index) {
                choice.setSelectedIndex(index);
            }
        });

        // Send events from Form to Widget
        choice.addListener(new ChoiceChangeListener() {
            public void onSetSelectedIndex (int index) {
                widget.selectIndex(index);
            }
        });
    }
}
```

```

    // Add children
    for (Form form : choice.getChildren())
        widget.addForm(form.apply(this));

    // Select initial choice
    widget.selectIndex(choice.getSelectedIndex());

    return widget;
}
}

```

First we create an instance of `ChoiceFormWidget` which will render the *Form Choice* and will receive an event in case the user selects a new item. By default this event is ignored.

We register the Widget with a listener which updates the *Form*.

Then we register the *Form* with a listener which updates the Widget.

After that we traverse all children and add the corresponding Widgets produced.

Finally we configure the Widget with the initial choice.

Notice that in case of an event the Widget does not update itself first. Instead it delegates the event to the *Form* and then receives another event from it. This ensures that all listeners (*XML Readers*, *XML Writers* etc) registered with the *Form* get the event.

The `ChoiceFormWidget` has the following methods:

```

class ChoiceFormWidget extends BaseUIWidget {

    interface ChoiceEventListener { void selectIndex(int index); }

    List<UiForm> children = new ArrayList<UiForm>();
    int selectedIndex;
    ChoiceEventListener listener;

    void addChild(UiForm form) {
        children.add(form);
    }
    void setListener(ChoiceEventListener listener) {
        this.listener = listener;
    }
    void selectIndex(int index) {
        selectedIndex = index;
    }

    int getSelectedIndex() {
        return selectedIndex;
    }
    UiForm getSelectedChild() {
        return children.get(selectedIndex);
    }

    void init() throws Exception {

```

```

        setViewSelector("form/choice");

        // Register child forms
        int i = 0;
        for (UiForm child : children)
            addWidget(String.valueOf(i++), child);
    }

    void handleEventSelect(String param) {
        listener.selectIndex(Integer.parseInt(param));
    }
    boolean saveAndValidate() {
        return getSelectedChild().saveAndValidate();
    }
}

```

`ChoiceFormWidget` has three fields which contain its children, the selected index and the listener. The first three methods enable to set these values.

`getSelectedIndex()` returns the select index and `getSelectedChild()` returns the selected child widget.

The `init()` and `handleEventSelect()` are the only Aranea-specific methods here.

`init()` selects the JSP and registers the child widgets.

When the user clicks on a button to select a new item `handleEventSelect()` is invoked. Instead of setting the `selectedIndex` value directly the event is delegated to the listener.

Finally `saveAndValidate()` is just delegated to the selected child widget.

The `ChoiceFormWidget` and `FormChoice` instances are isolated from each other as there are no explicit references. `AraneaFormBuilder` is the only one linking them together.

The rest of the *Form* components are handled similarly – the corresponding Widgets are not aware of their backend and vice versa. The two layers are tied using the `AraneaFormBuilder` methods which implement tiny listeners in either direction.

We continue by discussing the last topic about Aranea which covers how we convert our *Form Elements* into Aranea Controls.

3.4. Converting Form Element

Finally we observe how the *Form Elements* are converted into Aranea Widgets.

Aranea provides `org.araneaframework.uilib.form.FormWidget` for gathering user input, converting it to model objects and validating it. For each form field a control, a data object, possible constraints must be assigned. This is similar to the configuration of our *Form Elements* so we just need to match our API with the **Aranea Forms API**. To render the controls we can use Aranea JSP Tag `<ui:automaticFormElement/>` This enables to choose the actual JSP Tag to be rendered dynamically in the Widget code.

The conversion rules for *Controls* are listed in Table 2.

XML simple type	Form Control	Aranea Control	Aranea JSP Tag
string	<i>Input, Text</i>	TextControl	textInput
	<i>Secret</i>		passwordInput
	<i>TextArea</i>	TextareaControl	textarea
integer, long, int, short, byte, decimal	<i>Input</i>	NumberControl	numberInput
date		FloatControl	floatInput
time		DateControl	dateInput
datetime		TimeControl	timeInput
boolean		DateTimeControl	dateTimeInput
(any)		CheckboxControl	checkbox
(any)	<i>SelectOne, SelectBox</i>	SelectControl	select
	<i>RadioSelectBox</i>		radioSelect

Table 2. The conversion of Form Elements

Corresponding to the Aranea Forms API the controls are not used standalone. Instead they must be added to a `FormWidget` to form Aranea form elements. While converting the *Form* we first create a `FormWidget` per *Form Element*. To optimize the result we merge the consecutive ones to produce as few Widgets as possible.

We exclude the actual implementation of `UiWidget` as it is too complex.

This finishes the discussion about rendering form with Aranea web framework.

3.5. Embedding the Form

As we have stated earlier DynaForm is a **reusable web component** based on Aranea Web Framework.

We have described the `XmlFormBuilder` and `AraneaFormBuilder`. The first converts an `XSSchemaSet` into an `XmlForm`. The second takes a `Form` (obtained by invoking `xmlForm.getForm()`) and produces the Aranea Widget hierarchy.

So given an XML schema the corresponding Aranea Widget is gained using the following code snippet:

```
File file = new File("sample.xsd");
XmlForm xmlForm = SchemaUtil.toXmlForm(file);
Form form = xmlForm.getForm();
UiForm widget = AraneaFormBuilder.build(form);
```

First we assume that we have the schema file.

We invoke `SchemaUtil.toXmlForm()` to produce the `XmlForm`. The method first parses the file into an `XSSchemaSet` and then uses `XmlFormBuilder` to produce the result.

After that we extract the `Form` out of the `XmlForm`.

Finally we use `AraneaFormBuilder` to produce the widget. This can be registered as a child of any other `Widget`. The only restriction is that the latter must be able to call the `widget.saveAndValidate()` to validate the entered values and commit them to the *Form*.

Rendering the *Form* alone is useless as the program provides no data to the user and vice versa. Although it is possible to access the *Form* data using the *Form* API directly or by implementing a custom form visitor it is easier to use the data as an XML document.

To provide the *Form* with (initial) data *XmlReader* can be used:

```
XmlUtil.readXml(xmlString, xmlForm.getReader());
```

This populates the *Form* with data read from the given XML string.

To produce an XML string out of the *Form* *XmlWriter* can be used:

```
String xmlString = XmlUtil.writeXml(xmlForm.getWriter());
```

The traverses the whole *Form* and produces an XML based on the current values.

This enables the developer to embed a generated *Form* into his/her own `Widget` and manipulate with the *Form* data. In the last chapter we see how to use the `DynaData` API to customize the presentation of the *Form*.

4. DynaData

This is the final chapter in which we shortly describe the API of DynaData.

The generated *Form* stores the data in memory but it is not capable of reading or writing a DynaData file. Instead a **DynaData** instance is created by providing it with a *Form*. It always first scans the *Form* to read the default values. After that the *DynaForm* instance is able to read or write a DynaData file at any time.

The synchronization of the *Form* and DynaData file is achieved by the following code snippet:

```
Form form = ...
File file = ...
DynaData dynadata = new DynaDataImpl(form);
if (file.exists())
    dynadata.read(new FileReader(file));
dynadata.write(new FileWriter(file));

class DynaDataImpl implements DynaData {
    DynaDataImpl(Form form) { ... }
    ...
}

interface DynaData {
    void read(Reader in);
    void write(Writer out);
}
```

First we assume that we have the *Form* and DynaData file instances.

Then we create the `DynaDataImpl` instance by providing it the *Form*. This allows it to traverse the *Form* and read the default values. Besides the *Form* instance is stored to later populate it with new data. As *Form Repeats* repeat the same child component only the first child is traversed when the data is collected. In case a *Form Repeat* has no children a temporary child is automatically created.

As the DynaData file is optional we let `DynaDataImpl` to read it if it exists. When a file is read the *Form* is traversed hierarchically to populate it with new data. As *Form Repeat* components may later create additional children, listeners are registered to automatically update new children as well.

Finally we make `DynaDataImpl` to store the values to the file. At this moment it knows which form components were provided custom data, which were not and also which custom data could not be applied as there were no matching form components. According

to this distinguishing the DynaData file is divided into *Custom*, *Generated* and *Broken* sections as described earlier.

The implementation of `DynaDataImpl` is long and complex so we exclude it here.

This finishes the appendix about implementation details of DynaForm.

Appendix 2

CD with the Source Code

The CD contains source code of the DynaForm and its sample application.